PROGRAM

═══════════════

PROOF

Samuel MIMRAM

Contents

Introduction

These are the extended notes for the INF551 course which I taught at École Polytechnique starting from 2019. The goal is to give a first introduction to the Curry-Howard correspondence between programs and proofs from a theoretical programmer's perspective: we want to understand the theory behind logic and programming languages, but also to write concrete programs (in OCaml) and proofs (in Agda). Although most of the material is self-contained, the reader is supposed to be already acquainted with logic and programming.

0.1 Proving instead of testing

Most of the current software development is validated by performing tests: we run the programs with various values for the parameters, chosen in order to cover most branches of the program, and, if no bug has occurred during those executions, we consider that the program is good enough for production. The reason for this is that we consider that if the program uses "small" constants and "regular enough" functions then a large number of tests should be able to cover all the general behaviors of the program. Seriously following such a discipline greatly reduces the number of bugs, especially the simple ones, but we all know that it does not completely eliminates those: in some very particular and unlucky situations, problems still do happen.

In mathematics, the usual approach is quite different. For instance, when proving a property $P(n)$ over natural numbers, a typical mathematician will not test that $P(0)$ holds, $P(1)$ holds, $P(2)$ holds, and so on, up to a big number, and, if the property is experimentally always verified, claim: "I am almost certain that the property P is always true". He will maybe perform some tests in order to determine whether the conjecture is plausible or not, but in the end he will write down a proof, which ensures that the property $P(n)$ is always satisfied, for eternity, even if someone makes a particularly unlucky or perverse choice for n. Proving instead of testing does require some extra work, but the confidence it brings to the results is incomparable.

Let us present an extreme example of why this is the right way to proceed. On day one, our mathematician finds out using a formal computation software that

$$\int_0^\infty \frac{\sin(t)}{t} \mathrm{d}t = \frac{\pi}{2}$$

On day two, he tries to play around a bit with such formulas and finds out that

$$\int_0^\infty \frac{\sin(t)}{t} \frac{\sin(t/101)}{t/101} \mathrm{d}t = \frac{\pi}{2}$$

On day three, he thinks maybe a pattern could emerge and discovers that

$$\int_0^\infty \frac{\sin(t)}{t} \frac{\sin(t/101)}{t/101} \frac{\sin(t/201)}{t/201} \mathrm{d}t = \frac{\pi}{2}$$

On day four, he gets quite confident and conjectures that, for every $n \in \mathbb{N}$,

$$\int_0^\infty \left(\prod_{i=0}^n \frac{\sin(t/(100i+1))}{t/(100i+1)} \right) \mathrm{d}t = \frac{\pi}{2}$$

He then spends the rest of the year heating his computer and the planet, successfully proving the conjecture for increasing values of n. This approach seems to be justified since the most complicated function involved in here is the sine, which is quite regular (it is periodic), and all the constants are small (we get factors such as 100), so that if something bad ought to happen it will happen for a not-so-big value of n and testing should discover it. In fact, the conjecture breaks starting at

$$n = 15\ 341\ 178\ 777\ 673\ 149\ 429\ 167\ 740\ 440\ 969\ 249\ 338\ 310\ 889$$

and none of the usual tests would have found this out. There is a nice explanation for this which we will not give here, see [BB01, Bae18], but the moral is: if you want to be sure of something, don't test it, prove it.

On the computer science side, analogous examples abound where errors have been found in programs which were heavily tested. The number of such examples have recently increased with the advent of parallel computing (for instance, in order to exploit all the cores that you have on your laptop or even your smartphone), where bugs might be triggered by some particular and very rare scheduling of processes. Already in the 70s, Dijkstra was claiming that *program testing can be used to show the presence of bugs, but never to show their absence!* [Dij70], and the idea of formally verifying programs can even be traced back 20 years before that by, as usual, Turing [Tur49]. If we want to have software we can really trust (and not trust most of the time), we should move from testing to proving in computer science too.

In this course, you will learn how to perform such proofs, as well as the theory behind it. Actually, the most complicated program we will prove correct here is a sorting algorithm and I can already hear you thinking "come on, we have been writing sorting algorithms for decades, we should know how to write one by now". While I understand your point, I have two remarks to provide for this. Firstly, proving a more realistic program is only a matter of time (and experience): the course covers most of the concepts required to perform proofs, and attacking full-fledged code will not require new techniques, only patience. Secondly, in 2015, some researchers found out, using formal methods, that the default sorting algorithm (the one in the standard library, not some obscure library found on GitHub) in both Python and Java (not some obscure programming language) was flawed, and the bug had been there for more than a decade [dGdBB+19]...

0.2 Typing as proving

But how can we achieve this goal of applying techniques of proofs to programs? It turns out that we do not even need to come up with some new ideas, thanks to the so-called *proof-as-program correspondence* discovered in the 1960s by Curry and Howard: a program is secretly the same as a proof! More precisely, in a typed functional programming language, the type of a program can be read as

a formula, and the program itself contains exactly the information required to prove this formula. This is the one thing to remember from this course:

<div align="center">PROGRAM = PROOF</div>

This deep relationship allows the use of techniques from mathematics in order to study programs, but also can be used to extract computational contents from proofs in mathematics.

The goal of this course is to give precise meaning to this vague description, but let us give an example in order to understand it better. In a functional language such as OCaml, we can write a function such as

```
let comp f g x = g (f x)
```

and the compiler will automatically infer a type for it. Here, it will be

```
('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
```

meaning that for any types 'a, 'b and 'c,

- if f is a function which takes a value of type 'a as argument and returns a value of type 'b,

- if g is a function which takes a value of type 'b as argument and returns a value of type 'c, and

- if x is a value of type 'a,

then the result is of type 'c. For instance, with the function succ of type int -> int (it adds one to an integer), and the function string_of_int of type int -> string (it converts an integer to a string), the expression

```
comp succ string_of_int 2
```

will be of type string (it will evaluate to "3"). Now, if we read -> as a logical implication $\Rightarrow$, the type can be written as

$$(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$$

which is a valid formula. This is not by chance: in some sense, the program comp can be considered as a way of proving that this formula is valid.

Of course, if we want to prove richer properties of programs (or use programs to prove more interesting formulas), we should use a logic which is more expressive than propositional logic. In this course, we will present *dependent types* which achieve this, while keeping the proof-as-program correspondence. For instance, the euclidean division, which computes the quotient and remainder of two integers, is usually given the type

<div align="center">int -> int -> int * int</div>

stating that it takes two integers as arguments and returns a pair of integers. This typing is very weak, in the sense that there are many different functions which also have this type. With dependent types, we will be able to give it the type

$$(m : \mathtt{int}) \to (n : \mathtt{int'}) \to \Sigma(q : \mathtt{int}).\Sigma(r : \mathtt{int}).((m = nq + r) \times (0 \leqslant r < |n|))$$

which can be read as the formula

$$\forall m \in \texttt{int}.\forall n \in \texttt{int'}.\exists q \in \texttt{int}.\exists r \in \texttt{int}.((m = nq + r) \land (0 \leqslant r < |n|))$$

and entirely specifies its behavior (here, `int'` stands for the type of non-zero integers, division by zero being undefined).

0.3 Checking programs

In order to help formalize proofs with the aid of a computer people have developed *proof assistants* such as Agda, Coq or Lean: those are programs which help the user to gradually develop proofs (which is necessary due to their typical size) and automatically check that they are correct. While those have progressed much over the years, in practice, proving that a program is correct still takes much more time than testing it (but, again, the result is infinitely superior). For this reason, they have been used in areas where there is a strong incentive to do it: applications where human lives (or large amounts of money) are involved.

This technology is part of a bigger family of tools and techniques called *formal methods* whose aim is to guarantee the functioning of programs, with various automation levels and expressiveness. As usual, the more precise the invariants you are going to prove about your programs, the less automated you can be:
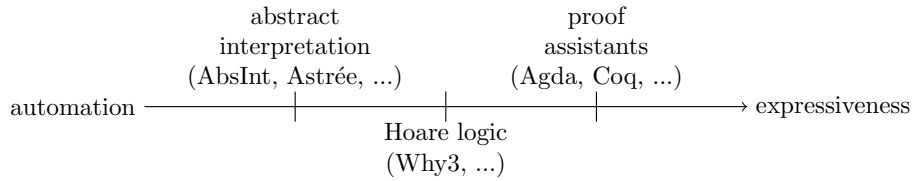
There is quite a number of industrial successes of uses of formal methods. For instance, the line 14 in Paris and the CDGVAL at Roissy airport have been proved using the B-method, Airbus is heavily using various formal tools (AbsInt, Astrée, CompCert, Frama-C), etc. We should also mention here the CompCert project, which provides a fully certified compiler for a (subset of) C: even though your program is proved to be bug-free, the compiler (which is not an easy piece of software) might itself be the cause of problems in your program...

The upside of automated methods is that, of course, they allow for reaching one's goal much more quickly. Their downside is that when they do not apply to a particular problem, one is left without a way out. On the other side, virtually any property can be shown in a proof assistant, provided that one is smart enough and has enough time to spend.

0.4 Checking proofs

Verifying that a proof is correct is a task which can be automatically and efficiently performed (it amounts to checking that a program is correctly typed); in contrast, finding a proof is an art. The situation is somewhat similar to analysis in mathematics, where differentiating a function is a completely mechanical

task, while integrating requires the use of many methods and tricks [Mun19]. This means that computer science can also be of some help to mathematicians: we can formalize mathematical proofs in proof assistants and ensure that no one has made a mistake. And it happens that mathematicians, even famous ones, make subtle mistakes.

For instance, in the 1990s, the Fields medalist Voevodsky wrote a paper solving an important conjecture by Grothendieck [KV91], which roughly states that spaces are the same as strict ∞-categories in which all morphisms are weakly invertible (don't worry if you do not precisely understand all the terms in this sentence). A few years later, this was shown to be wrong because someone provided a counter-example [Sim98], but no one could exactly point out what was the mistake in the original proof. Because of this, Voevodsky thought for more than 20 years (i.e. even after the counter-example was found) that his proof was still correct. Understanding that there was indeed a mistake lead him to use proof assistants for all his proofs and, in fact, propose a new foundation for mathematics using logics, which is nowadays called *homotopy type theory* [Uni13]. Quoting him [Voe14]:

> *I now do my mathematics with a proof assistant and do not have to worry all the time about mistakes in my arguments or about how to convince others that my arguments are correct.*
>
> *But I think that the sense of urgency that pushed me to hurry with the program remains. Sooner or later computer proof assistants will become the norm, but the longer this process takes the more misery associated with mistakes and with unnecessary self-verification the practitioners of the field will have to endure.*

As a much simpler example, suppose that we want to prove that all horses have the same color (sic). We show by induction on n the property $P(n) =$ "every set of n horses is monochromatic". For $n = 0$ and $n = 1$, the property is obvious. Now suppose that $P(n)$ holds and consider a set H of $n+1$ horses. We can figure H as a big set, in which we can pick two distinct elements (horses) h_1 and h_2 and consider the sets $H_1 = H \setminus \{h_2\}$ and $H_2 = H \setminus \{h_1\}$:
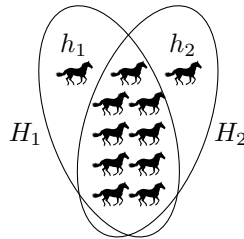
By induction hypothesis all the horses in H_1 have the same color and all the horses in H_2 have the same color. Therefore, by transitivity, all the horses in H have the same color. Of course this proof is not valid, because we all know that there are horses of various colors (can you spot the mistake?). Formalizing the proof in a proof assistant will force you to fill in all the details, thus removing the possibility for potential errors in vague arguments, and will ensure that the arguments given are actually valid, so that flaws such as in the above proof will be uncovered. This is not limited to small proofs: large and important

proofs have been fully checked, such as the four color theorem (in Coq) in graph theory [Gon08], the Feit-Thompson theorem (in Coq) which is central in the classification of finite simple groups [GAA$^+$13], a proof of the Kepler conjecture on dense sphere packing (in HOL light and Isabelle) [HAB$^+$17], or results from condensed mathematics (the "liquid tensor experiment", in Lean) [Sch22].

0.5 Searching for proofs

Closely related to proof checking is *proof search*, or *automated theorem proving*, i.e. have the computer try by itself to find a proof for a given formula. For simple enough fragments of logic (e.g. propositional logic) this can be done: proof theory allows to carefully design efficient new proof search procedures. For richer logics, it quickly becomes undecidable: it was first hoped that we could algorithmically decide whether an arbitrary formula is provable or not, this is Hilbert and Ackermann's *Entscheidungsproblem* formulated in 1928 [HA28], but this was shown not to be the case in 1936 independently by Church [Chu36b, Chu36a] and Turing [Tur37a], respectively using λ-terms and Turing machines as formalism for expressing algorithms. Still, modern proof assistants (e.g. Coq or Lean) have so called *tactics* which can fill in some specific proofs, even though the logic is rich. For example, they are able to take care of showing boring identities such as $(x + y) - x = y$ in abelian groups.

Understanding proof theory allows us to formulate problems in a logical fashion and solve them. It thus applies to various fields, even outside theoretical computer science. For instance, McCarthy, a founder of Artificial Intelligence (the name is due to him!), was a strong advocate of using mathematical logic to represent knowledge and manipulate it [McC60]. Neural networks are admittedly more fashionable these days, but one never knows what the future will be made of.

Although we will see some proof search techniques in this course, this will not be a central subject. The reason for this is that the main message is that we should take proofs seriously: since a proof is the same as a program, we are not interested in provability, but rather in proofs themselves, and proof search techniques give us very little control over the proofs they produce.

0.6 Foundations

At the beginning of the 20th century, some annoying paradoxes surfaced in mathematics, such as Russell's paradox, motivating Hilbert's program to provide an axiomatization on which all mathematics could be founded and show that this axiomatization is consistent: this is sometimes called the *foundational crisis*. Although Gödel's incompleteness theorems established that there is no definite answer to this question, various formalisms have been elaborated, in which one can develop most of usual mathematics. One of the most widely used is set theory, as axiomatized by Zermelo and Fraenkel, but other formalisms have been proposed, such as Russell's *theory of types* [WR12], where the modern type theory originates from: in fact, type theory can be taken as a foundation of mathematics. People usually see set theory as being more fundamental, since we see a type as representing a set (e.g. $A \Rightarrow B$ is the set of functions from

the set of *A* to the one of *B*), but we can also view type theory as being more fundamental since we can formalize set theory in type theory. The one you take for foundations is a matter of taste: are you more into chickens or into eggs?

Type theory also provides a solid framework in which one can study basic philosophical questions such as: What is reasoning? What is a proof? If I know that something exists, do I know this thing? What does it mean for two things to be equal? and so on. We could spend pages discussing those matters (and others have done so), but we rather like to formalize things, and we will see that very satisfactory answers to those questions can be given with a few inference rules. The meaning of life remains an open question, though.

By taking an increasingly important part in our lives and influencing the way we see the (mathematical) world, these ideas even have evolved for some of us into some sort of religion based on the *computational trinitarism*, which stems from the observation that computation manifests itself in three forms [Har11]:

$$
\begin{array}{c}
\text{categories} \\
\diagup \quad \diagdown \\
\text{logic} \longrightarrow \text{programming}
\end{array}
$$
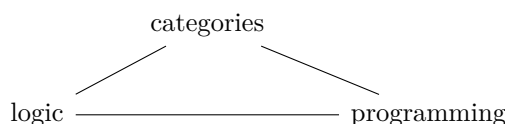
The aim of the present text is to explain the bottom line of the above diagram and leave categories for other books [Mac71, LS88, Jac99]. Another closely related religion is *constructivism*, a doctrine according to which something can be accepted only if it can actually be constructed. It will play a central role in here, because programs precisely constitute a mean to describe the construction of things.

0.7 In this course

As a first introduction to functional and typed languages, we first present OCaml in chapter 1, in which most example programs given here are written. We present propositional logic in chapter 2 (the proofs), λ-calculus in chapter 3 (the programs), and the simply-typed variant λ-calculus in chapter 4 (the programs are the proofs). We then generalize the correspondence between proofs and programs to richer logics: we present first-order logic in chapter 5, and, in chapter 6, the proof assistant Agda, which is used in chapter 7 to formalize most important results in this book, and is based on the dependent types detailed in chapter 8. We finally give an introduction to the recent developments in homotopy type theory in chapter 9.

0.8 Other references on programs and proofs

Although we claim some originality in the treatment and the covered topics, this book is certainly not the first one about the subject. Excellent references include Girard's *Proofs and Types* [Gir89], Girard's *Blind Spot* [Gir11], Leroy's Collège de France course *Programmer = démontrer ? La correspondance de Curry-Howard aujourd'hui*, Pierce's *Types and Programming Languages* [Pie02], Sørensen and Urzyczyn's *Lectures on the Curry-Howard isomorphism* [SU06],

the "HoTT book" *Homotopy Type Theory: Univalent Foundations of Mathematics* [Uni13], *Programming Language Foundations in Agda* [WK19] and *Software foundations* [PdAC⁺10].

0.9 About this document

This book was first published in 2020, and the version you are currently reading was compiled on April 3, 2024. Regular revisions can be expected if mistakes are found. Should you find one, please send me a mail at the address `samuel.mimram@lix.polytechnique.fr`.

Reading on the beach. A printed copy of this course can be ordered from Amazon: `https://www.amazon.com/dp/B08C97TD9G/`.

Color of the cover. In case you wonder, the color of the cover was chosen because it seemed obvious to me that

$$\text{program} = \text{proof} = \text{purple}$$

Code snippets. Most of the code shown in this book is excerpted from larger files which are regularly compiled in order to ensure their correctness. The process of extracting snippets for inclusion into LaTeX is automated with a tool whose code is freely available at `https://github.com/smimram/snippetor`.

Thanks. Many people have (knowingly or not) contributed to the development of this book. I would like to particularly express my thanks to David Baelde, Olivier Bournez, Eric Finster, Emmanuel Haucourt, Daniel Hirschkoff, Stéphane Lengrand, Assia Mahboubi, Paul-André Melliès, Gabriel Scherer, Pierre-Yves Strub, Benjamin Werner.

I would also like to express my thanks to the readers of the book who have suggested corrections and improvements: Eduardo Jorge Barbosa, Brian Berns, Alve Björk, Florian Chudigiewitsch, Adam Dingle, Aran Donohue, Maximilian Doré, Leonid Dubinsky, Sylvain Henry, Chhi'mèd Künzang, Yuxi Liu, Jeremy Roach, Kyle Stemen, Marc Sunet, Kenton Van, Yuval Wyborski, Uma Zalakain.

Typed functional programming

1.1 Introduction

As an illustration of typed functional programming, we present here the OCaml programming language, which was developed by Leroy and collaborators, following ideas from Milner [Mil78]. We present here some of the basics of the language both because it will be used in order to provide illustrative implementations, and also because we will detail the theory behind it and generalize it in later chapters. This is not meant to be a complete introduction to programming in OCaml: advanced courses and documentation can be found on the website `http://ocaml.org/`, as well as in books [CMP00, MMH13].

After a brief tour of the language, we present the most important constructions in section 1.2, and detail recursive types, which are the main way of constructing types throughout the book, in section 1.3. In section 1.4, we present the ideas behind the typing system and the guarantees it brings. Finally, we illustrate how types can be thought of as formulas in section 1.5.

1.1.1 Hello world. The mandatory "Hello world!" program, which prints `Hello world!`, can be written as follows:

```
(* Our first program. *)
print_endline "Hello, world!"
```

This illustrates the concise syntax of the language (compared to Java for instance). Comments are written using `(* ... *)`. Application of a function to arguments does not require parenthesis. The indentation is not relevant in programs (unlike e.g. Python), but you are of course strongly encouraged to indent your programs nicely.

1.1.2 Execution. The programs written in OCaml can be compiled to efficient native code by using `ocamlopt`, but there is also a "toplevel" which allows to interactively evaluate commands. It can be launched by typing `ocaml`, or `utop` if you want a fancier version. For instance:

```
# 2 + 2;;
- : int = 4
```

We have typed `2 + 2`, followed by `;;` to indicate the end of our program. The toplevel then indicates that this is an integer (it is of type `int`) and that the result is `4`. We call *value* an expression which cannot be reduced further: `2+2` is not a value, whereas `4` is: the execution of a program consists in reducing expressions to values in an appropriate way (e.g. `2+2` reduces to `4`).

1.1.3 A statically typed language. The OCaml language is *typed*, meaning that every term has a type indicating the kind of data it is. A type can be thought of as a particular set of values, e.g. `int` represents the set of integers, `string` represents the set of strings, and so on. In this way, the expressions `2+2` and `4` have the type `int` (they are integers), and the function `string_of_int` which gives the string representation of an integer has type `int -> string`, meaning that it is a function which takes an integer as argument and returns a string. Moreover, typing is *statically* checked: when compiling a program, the compiler ensures that all the types match, and we use values of the expected type. For instance, if we try to compile the program

```
let s = string_of_int 3.2
```

the compiler will complain with

```
Error: This expression has type float but an expression was
       expected of type int
```

because the `string_of_int` function expects an integer whereas we have provided a float number as argument. This discipline is very strict in OCaml (we sometimes say that the typing is *strong*): this ensures that the program will not raise an error during execution because an unexpected value was provided to a function (this is a theorem!). In other words, quoting Milner [Mil78]:

> *Well-typed programs cannot go wrong.*

Moreover, the types are *inferred*, meaning that the user never has to specify the types, they are guessed automatically. For instance, in the definition

```
let f x = x + 1
```

we know that the addition takes two integers as arguments and returns an integer: therefore x must be of type `int` and the compiler infers that `f` must be a function of type `int -> int`. However, if for some reason we want to specify the types, it is still possible:

```
let f (x : int) : int = x + 1
```

1.1.4 A functional language. The language is *functional*, meaning that it has good support for defining functions and manipulating them just as any other value. For instance, suppose that we have a list `l` of integers, and we want to double all its elements. We can use the `List.map` function from the standard library, which is of type

```
('a -> 'b) -> 'a list -> 'b list
```

meaning that it takes as arguments a function f of type `'a -> 'b` (here `'a` and `'b` are intended to mean "any type"), and a list whose elements are of type `'a`, and returns the list whose elements are of type `'b`, obtained by applying f to all the elements of the list. We can then define the "doubled list" by

```
let l2 = List.map (fun x -> 2 * x) l
```

where we apply the function $x \mapsto 2 \times x$ to every element: note that the function `List.map` takes a function as argument and that we did not even have to give a name to the function above by using the construction `fun` (such a function is sometimes called *anonymous*).

1.1.5 Other features. There are some other important features of the OCaml language that we mention only briefly here, because we will not use them much.

References. As in most programming languages, OCaml has support for values that we can modify, here called *references*: they can be thought of as memory cells, from which we can retrieve the value and also change it. We can create a reference `r` containing `x` with `let r = ref x`, then we can obtain its contents with `!r` and change it to `y` with `r := y`. For instance, incrementing a counter 10 times is done by

```
let () =
  let r = ref 0 in
  for i = 0 to 9 do
    r := !r + 1
  done
```

Garbage collection. Unlike languages such as C, OCaml has a *Garbage Collector* which takes care of allocating and freeing memory. This means that freeing memory for a value which is not used anymore is taken care of for us. This prevents many common bugs such as writing in a part of memory which was freed or freeing a memory region twice by mistake.

Other traits. In addition to the functional programming style, OCaml has support for many other styles of programming including *imperative* (e.g. references described above), *objects*, etc. OCaml also has support for *records*, *arrays*, *modules*, *generalized algebraic data types*, etc.

1.2 Basic constructions

1.2.1 Declarations. Values are declared with the `let` keyword. For instance,

```
let x = 3
```

declares that the variable `x` refers to the value `3`. Note that, unless we explicitly use the reference mechanism, these values cannot be modified. An expression can also contain some local definitions which are only visible in the rest of the expression. For instance, in

```
let x =
  let y = 2 in
  y * y
```

the definition of the variable `y` is only valid in the following expression `y * y`.

A program consists in a sequence of such definitions. In the case where a function "does not return anything", such as printing, by convention it returns a value of type `unit`, and we often use the construction `let () = ...` in order to ensure that this is the case. For instance:

```
let x = "hello"

let () = print_string ("The value of x is " ^ x)
```

1.2.2 Functions. Functions are also defined using `let` definition, specifying the arguments after the name of the variable. For instance,

```
let add x y = x + y
```

which would be of type

```
int -> int -> int
```

Note that arrows are implicitly bracketed to the right: this type means

```
int -> (int -> int)
```

Application of a function to arguments is written as the juxtaposition of the function and the arguments, e.g.

```
let x = add 3 4
```

(no need for parenthesis). *Partial application* is supported, meaning that we do not have to give all the arguments to a function (functions are sometimes called *curried*). For instance, the incrementing of an integer can be defined by

```
let incr = add 1
```

The value `incr` thus defined is the function which takes an argument `y` and returns `add 1 y`, so that the above definition is equivalent to

```
let incr y = add 1 y
```

This is in accordance to the bracketing of the types above: `add` is a function which, when given an integer argument, returns a function of type `int -> int`.

As mentioned above, anonymous functions can be defined by the construction `fun x -> ...`. The `add` function could thus have been equivalently defined by

```
let add = fun x y -> x + y
```

or even

```
let add x = fun y -> x + y
```

Functions can be recursive, meaning that they can call themselves. In this case, the `rec` keyword has to be used. For instance, the factorial function is defined by

```
let rec fact n =
  if n = 0 then 1 else n * fact (n - 1)
```

It is possible to define two mutually recursive functions `f` and `g` by using the following syntax:

```
let rec f x = ...
and     g x = ...
```

This means that we can use both `f` and `g` in the definitions of `f` and `g` (see figure 1.4 below for an example).

1.2.3 Booleans. The type corresponding to booleans is `bool`, its two values being `true` and `false`. The usual operators are present: conjunction `&&`, disjunction `||`, and negation `not`. In order to test whether two values `x` and `y` are equal or different, one can use `x = y` and `x <> y`. They can be used in conditional branchings

```
if ... then ... else ...
```

or loops

```
while ... do ... done
```

Beware that the operators `==` and `!=` also exist, but they compare values physically, i.e. check whether they have the same memory location, not if they have the same contents. For instance, using the toplevel, we have:

```
# let x = ref 0;;
val x : int ref = {contents = 0}
# let y = ref 0;;
val y : int ref = {contents = 0}
# x = x;;
- : bool = true
# x = y;;
- : bool = true
# x == x;;
- : bool = true
# x == y;;
- : bool = false
```

1.2.4 Products. The *pair* of `x` and `y` is written `x,y`. For instance, we can consider the pair `3,"hello"` which has the *product* type `int * string` (it is a pair consisting of an integer and a string). Note that addition could have been defined as

```
let add' (x,y) = x + y
```

resulting in a slightly different function than above: it now has the type

```
(int * int) -> int
```

meaning that it takes one argument, which is a pair of integers, and returns an integer. This means that partial application is not directly available as before, although we could still write

```
let incr' = fun y -> add (1,y)
```

1.2.5 Lists. We quite often use *lists* in OCaml. The empty list is `[]`, and `x::l` is the list obtained by putting the value `x` before a list `l`. Most expected functions on lists are available in the module `List`. For instance,

- `@` concatenates two lists,

- `List.length` computes the length of a list,

- `List.map` applies a function to all the elements of a list,

- `List.iter` executes a function for all the elements of a list,

- `List.mem` tests whether a value belongs to a list.

1.2.6 Strings. Strings are written as `"this is a string"` and the related functions can be found in the module `String`. For instance, the function `String.length` computes the length of a string and `String.sub` computes a substring (at given indices) of a string. Concatenation is obtained by `^`.

1.2.7 Unit. In OCaml, the type `unit` contains only one element, written `()`. As explained above, this is the value returned by functions which only have an effect and return no meaningful value (e.g. printing a string). They are also quite useful as an argument for functions which have an effect. For instance, if we define

```
let f = print_string "hello"
```

the program will write "hello" at the beginning of the execution, because the expression defining `f` is evaluated. However, if we define

```
let f () = print_string "hello"
```

nothing will be printed because we define a function taking a `unit` as argument. In the course of the program, we can then use `f ()` in order to print "hello".

1.3 Recursive types

A very useful way of defining new data types in OCaml is by *recursive types*, whose elements are constructed from other types using specific constructions, called *constructors*.

1.3.1 Trees. As a first example, consider *trees* (more specifically, planar binary trees with integer labels) such as

$$
\begin{array}{c}
3 \\
\swarrow \qquad \searrow \\
4 \qquad\qquad 1 \\
\swarrow \quad \searrow \\
1 \qquad 3 \\
\swarrow \quad \searrow \\
5 \qquad 2
\end{array}
$$
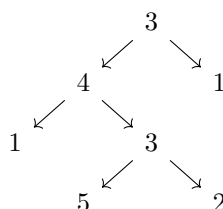
Here, a tree consists of finitely many nodes which are labeled by an integer and can either have two children, which are themselves trees, or none (in which case they are called *leaves*). This description translates immediately to the following type definition in OCaml:

```
type tree =
  | Node of int * tree * tree
  | Leaf of int
```

This says that a tree is recursively characterized as being `Node` applied to a triple consisting of an integer and two trees or `Leaf` applied to an integer. For instance, the above tree is represented as

```
let t = Node (3, Node (4, Leaf 1, Node (3, Leaf 5, Leaf 2)), Leaf 1)
```

Here, `Node` and `Leaf` are not functions (`Leaf 1` does not reduce to anything), they are called *constructors*. By convention, constructor names have to begin with a capital letter, in order to distinguish them from function names (which have to begin with a lowercase letter).

Pattern matching. Any element of the type `tree` is obtained as a constructor applied to some arguments. OCaml provides the `match` construction which allows to distinguish between the various possible cases of constructors and return a result accordingly: this is called *pattern matching*. For instance, the function computing the height of a tree can be implemented as

```
let rec height t =
  match t with
  | Node (n, t1, t2) -> 1 + max (height t1) (height t2)
  | Leaf n -> 0
```

Here, `Node (n, t1, t2)` is called a *pattern* and n, t1 and t2 are variables which will be defined as the values corresponding to the tree we are currently matching with, and could be given any other names. As another example, the sum of the labels in a tree can be computed as

```
let rec sum t =
  match t with
  | Node (n, t1, t2) -> n + sum t1 + sum t2
  | Leaf n -> n
```

It is sometimes useful to add conditions to matching cases, which can be done using the `when` construction. For instance, if we wanted to match only nodes with strictly positive labels, we could have used, in our pattern matching, a case of the form

```
  | Node (n, t1, t2) when n > 0 -> ...
```

In case where multiple cases match, the first one is chosen. OCaml tests that all the possible values are handled in a pattern matching (and issues a warning otherwise). Finally, one can write

```
let f = function ...
```

instead of

```
let f x = match x with ...
```

(this shortcut was introduced because it is very common to directly match on the argument of a function).

1.3.2 Usual recursive types. It is interesting to note that many (most) usual types can be encoded as recursive types.

Booleans. The type of booleans can be encoded as

```
type bool = True | False
```

although OCaml chose not to do that for performance reasons. A case construction

```
if b then e1 else e2
```

could then be encoded as

```
match b with
  | True  -> e1
  | False -> e2
```

Lists. Lists are also a recursive type:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

In OCaml, `[]` is a notation for `Nil` and `x::l` a notation for `Cons (x, l)`. The length of a list can be computed as

```
let rec length l =
  match l with
  | x::l -> 1 + length l
  | []   -> 0
```

Note that the type of list is parametrized over a type `'a`. We are thus able to define, at once, the type of lists containing elements of type `'a`, for any type `'a`.

Coproducts. We have seen that the elements of a product type `'a * 'b` are pairs `x , y` consisting of an element x of type `'a` and an element y of type `'b`. We can define *coproducts* consisting of an element of type `'a` *or* an element of type `'b` by

```
type ('a, 'b) coprod =
  | Left  of 'a
  | Right of 'b
```

An element of this type is of the form `Left x` with x of type `'a` or `Right y` with y of type `'b`. For instance, we can define a function which provides the string representation of a value which is either an integer or a float by

```
let to_string = function
  | Left  n -> string_of_int n
  | Right x -> string_of_float x
```

which is of type `(int, float) coprod -> string`.

Unit. The type `unit` has `()` as the only value. It could have been defined as

```
type unit =
  | T
```

having `()` being a notation for `T`.

Empty type. The "empty type" can be defined as

```
type empty = |
```

i.e. a recursive type with no constructor (we still need to write | for syntactical reasons). There is thus no values of that type.

Natural numbers. Natural numbers (in unary notation) can be defined as

```
type nat =
  | Zero
  | Suc of nat
```

(any natural number is either 0 or the successor of a natural number) and addition as

```
let rec add m n =
  match m with
  | Zero  -> n
  | Suc m -> Suc (add m n)
```

Of course, it would be a bad idea to use this type for heavy computations, and `int` provides access to machine binary integers (and thus natural numbers), which are much more efficient.

1.3.3 Abstract description. As indicated before, a type can be thought of as a set of values. We would now like to briefly sketch a mathematical definition of the set of values corresponding to inductive types.

Suppose fixed a set $\mathcal{U}$, which we can think of as the set of all possible values an OCaml program can manipulate. We write $\mathcal{P}(\mathcal{U})$ for the *powerset* of $\mathcal{U}$, i.e. the set of all subsets of $\mathcal{U}$, which is ordered by inclusion. Any recursive definition induces a function $F : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ sending a set X to the set obtained by applying the constructors to the elements of X. For instance, with the definition `tree` of section 1.3.1, the induced function is

$$F(X) = \{\mathtt{Node}(n, t_1, t_2) \mid n \in \mathbb{N} \text{ and } t_1, t_2 \in X\} \cup \{\mathtt{Leaf}(n) \mid n \in \mathbb{N}\}$$

The set associated to `tree` is intuitively the smallest set $X \subseteq \mathcal{U}$ which is closed under adding nodes and leaves, i.e. such that $F(X) = X$, provided that such a set exists. Such a set X satisfying $F(X) = X$ is called a *fixpoint* of F.

In order to be able to interpret the type of trees as the smallest fixpoint of F, we should first show that such a fixpoint indeed exists. A crucial observation in order to do so is the fact that the function $F : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ is *monotone*, in the sense that, for $X, Y \in \mathcal{U}$,

$$X \subseteq Y \text{ implies } F(X) \subseteq F(Y).$$

Theorem 1.3.3.1 (Knaster-Tarski [Kna28, Tar55]). The set

$$\mathrm{fix}(F) = \bigcap \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$$

is the least fixpoint of F: we have

$$F(\mathrm{fix}(F)) = \mathrm{fix}(F)$$

and, for every fixpoint X of F,

$$\text{fix}(F) \subseteq X$$

Proof. We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$ for the set of *prefixpoints* of F. Given $X \in \mathcal{C}$, we have

$$\text{fix}(F) = \bigcap \mathcal{C} \subseteq X \tag{1.1}$$

and therefore, since F is increasing,

$$F(\text{fix}(F)) \subseteq F(X) \subseteq X \tag{1.2}$$

Since this holds for any $X \in \mathcal{C}$, we have

$$F(\text{fix}(F)) \subseteq \bigcap \mathcal{C} = \text{fix}(F) \tag{1.3}$$

Moreover, by monotonicity again, we have

$$F(F(\text{fix}(F))) \subseteq F(\text{fix}(F))$$

therefore, $F(\text{fix}(F)) \in \mathcal{C}$, and thus by (1.1)

$$\text{fix}(F) \subseteq F(\text{fix}(F)) \tag{1.4}$$

From (1.3) and (1.4), we deduce that $\text{fix}(F)$ is a fixpoint of F. An arbitrary fixpoint X of F necessarily belongs to $\mathcal{C}$ and, by (1.2), we have

$$\text{fix}(F) = F(\text{fix}(F)) \subseteq X$$

$\text{fix}(F)$ is thus the smallest fixpoint of F. □

Remark 1.3.3.2. The attentive reader will have noticed that all we really used in the course of the proof was the fact that $\mathcal{P}(\mathcal{U})$ is a complete semilattice, i.e. we can compute arbitrary intersections. Under the more subtle hypothesis of the *Kleene fixpoint theorem* ($\mathcal{P}(\mathcal{U})$ is a directed complete partial order and F is Scott-continuous), one can even show that

$$\text{fix}(F) = \bigcup_{n \in \mathbb{N}} F^n(\emptyset)$$

i.e. the fixpoint can be obtained by iterating F from the empty set. In the case of trees,

$$F^0(\emptyset) = \emptyset$$
$$F^1(\emptyset) = \{\texttt{Leaf}(n) \mid n \in \mathbb{N}\}$$
$$F^2(\emptyset) = \{\texttt{Leaf}(n) \mid n \in \mathbb{N}\} \cup \{\texttt{Nodes}(n, t_1, t_2) \mid n \in \mathbb{N} \text{ and } t_1, t_2 \in F^1(\emptyset)\}$$

and more generally, $F^n(\emptyset)$ is the set of trees of height strictly below n. The theorem states that any tree is a tree of some (finite) height.

Remark 1.3.3.3. In general, there are multiple fixpoints. For instance, for the function F corresponding to trees, the set of all "trees" where we allow to have an infinite number of nodes is also a fixpoint of F.

As a direct corollary of theorem 1.3.3.1, we obtain the following *induction principle*:

Corollary 1.3.3.4. Given a set X such that $F(X) \subseteq X$, we have $\text{fix}(F) \subseteq X$.

Example 1.3.3.5. With the type `nat` of natural numbers, we have

$$F(X) = \{\text{Zero}\} \cup \{\text{Suc}(n) \mid n \in X\}$$

We have

$$\text{fix}(F) = \{\text{Suc}^n(\text{Zero}) \mid n \in \mathbb{N}\} = \{\text{Zero}, \text{Suc}(\text{Zero}), \text{Suc}(\text{Suc}(\text{Zero})), \ldots\}$$

In the following, we write 0 (resp. $S\,n$) instead of `Zero` (resp. $\text{Succ}(n)$), and $\text{fix}(F) = \mathbb{N}$. The induction principle states that if X contains 0 and is closed under successor, then it contains all natural numbers. Given a property $P(n)$ on natural numbers, consider the set

$$X = \{n \in \mathbb{N} \mid P(n)\}$$

The requirement $F(X) \subseteq X$ translates as $P(0)$ holds and $P(n)$ implies $P(S\,n)$. The induction principle is thus the classical induction principle for natural numbers:

$$P(0) \Rightarrow (\forall n \in \mathbb{N}.P(n) \Rightarrow P(S\,n)) \Rightarrow (\forall n \in \mathbb{N}.P(n))$$

Example 1.3.3.6. Consider the type `empty`. We have $F(X) = \emptyset$ and thus $\text{fix}(F) = \emptyset$. The induction principle states that any property is necessarily valid on all the elements of the empty set:

$$\forall x \in \emptyset.P(x)$$

Exercise 1.3.3.7. Define the function F associated to the type of lists. Show that it also has a greatest fixpoint, distinct from the smallest fixpoint, and provide a concrete description of it.

1.3.4 Option types and exceptions. Another quite useful recursive type defined in the standard library is the *option* type

```
type 'a option =
  | Some of 'a
  | None
```

A value of this type is either of the form `Some x` for some `x` of type `'a` or `None`. It can be thought of as the type `'a` extended with the default value `None` and can be used for functions that, in some cases, do not return a value (in other languages such as C or Java, one would return a NULL pointer in this case). For instance, the function returning the head of a list is almost always defined, except when the argument is the empty list. It thus makes sense to implement it as the function of type `'a list -> 'a option` defined by

```
let hd l =
  match l with
  | x::l -> Some x
  | []   -> None
```

It is however quite cumbersome to use, because each time we want to use the result of this function, we have to match it in order to decide whether the result is None or not. For instance, in order to double the head of a list l of integers known to be non-empty, we still have to write something like

```
match head l with
  | Some n -> 2*n
  | None   -> 0   (* This case cannot happen *)
```

See figure 1.2 for a more representative example.

Exceptions. In order to address this, OCaml provides the mechanism of *exceptions*, which are kinds of errors that can be raised and caught. For instance, in the standard library, the exception Not_found is defined by

```
exception Not_found
```

and the head function by

```
let hd l =
  match l with
  | x::l -> x
  | []   -> raise Not_found
```

It now has type 'a list -> 'a, meaning that we can write

```
2 * (hd l)
```

to double the head of a list l. In the case where we take the head of the empty list, the exception Not_found is raised. We can catch it with the following construction if we need to:

```
try
  ...
with
| Not_found -> ...
```

1.4 The typing system

We have already explained in section 1.1.3 that OCaml is a strongly typed language. We detail here some of the advantages and properties of such a typing system.

1.4.1 Usefulness of typing. One of the main advantages of typed languages is that typing ensures their safety: if the program passes the typechecking phase, we are guaranteed that we will not have errors during execution caused by unexpected data provided to a function, see section 1.4.3. But there are other advantages too.

Documentation. Knowing the type of a function is very useful for *documenting* it: from the type we can generally deduce the order of the arguments of the functions, what it is returning and so on. For instance, the function `Queue.add` of the module implementing queues in OCaml has type

```
'a -> 'a queue -> unit
```

This allows us to conclude that the function takes two arguments: the first argument must be the element we want to add and the second one the queue in which we want to add it. Finally, the function does not return anything (to be more precise it returns the only value of the type `unit`): this must mean that the structure of queue is modified in place (otherwise, we would have had the modified queue as return value).

Abstraction. Having a typing system is also good for *abstraction*: we can use a data structure without knowing the details of implementation or even having access to them. Taking the `Queue.add` function as example again, we only know that the second argument is of type `'a queue`, without any more details on this type. This means that we cannot mess up with the internals of the data structure, and that the implementation of queues can be radically modified without us having to change our code.

Efficiency. Static typing can also be used to improve efficiency of compiled programs. Namely, since we know in advance the type of the values we are going to handle, our code can be specific to the corresponding data structure, and avoid performing some security checks. For instance, in OCaml, the concatenation function on strings can simply put the two strings together; in contrast, in a dynamically typed programming language such as Python, the concatenation function on strings has first to ensure that the arguments are strings, if they are not we will try to convert them as strings, and then we can put them together.

1.4.2 Properties of typing. There are various flavors of typing systems.

Dynamic vs static. The types of programs can either be checked during the execution (the typing is *dynamic*) or during the compilation (the typing is *static*): OCaml is using the latter. Static typing has many advantages: potential errors are found very early, without having to perform tests, it can help to optimize programs, and provides very strong guarantees on the execution of the program. The dynamic approach also has some advantages though: the code is more flexible, the runtime can automatically perform conversions between types if necessary, etc.

Weak vs strong. The typing system of OCaml is *strong* which means that it ensures that the values in a type are actually of that type: there is no implicit or dynamic type conversion, no NULL pointers, no explicit manipulation of pointers, and so on. By opposition, when those requirements are not met, the typing system is said to be *weak*.

Decidability of typing. A basic requirement of a typing system is that we should be able to decide whether a given term has a given type, i.e. we should have a *type checking* algorithm. For OCaml (and all decent programming languages) this is the case, and type checking is performed during each compilation of a program.

Type inference. It is often cumbersome to have to specify the type of all the terms (or even to give many type annotations). In OCaml, the compiler performs *type inference*, which means that it automatically finds a type for the program, when there is one.

Polymorphism. The types in OCaml are *polymorphic*, which means that they can contain variables which are treated as universally quantified. For instance, the identity function

```
let id x = x
```

has the type `'a -> 'a`, which can also be read as the universally quantified type $\forall A.A \to A$. This means that we can substitute `'a` for any type and still get a valid type for the identity.

Principal types. A program can admit multiple types. For instance, the identity function admits the following types

```
  'a -> 'a      or      int -> int      or      ('a -> 'b) -> ('a -> 'b)
```

and infinitely many others. The first one `'a -> 'a` is however "more general" than the others because all the other types can be obtained by substituting `'a` by some type. Such a most general type is called a *principal type*. The type inference of OCaml has the property that it always produces a principal type.

1.4.3 Safety. The programs which are well-typed in OCaml are *safe* in the sense that types are preserved during execution and programs do not get stuck. In order to formalize these properties, we first need to introduce a notion of reduction, which formalizes the way programs are executed. We will first do this on a very small (but representative) subset of the language. Most of the concepts used here, such as reduction or typing derivation, will be further detailed in subsequent chapters.

A small language. We first introduce a small programming language, which can be considered as a very restricted subset of OCaml, that we will implement in OCaml itself. In this language, the *values* we use are either integers or booleans. We define a program as being either a value, an addition (`p + p'`), a comparison of integers (`p < p'`), or a conditional branching (`if p then p' else p"`):

```
type prog =
  | Bool of bool
  | Int  of int
  | Add  of prog * prog
  | Lt   of prog * prog
  | If   of prog * prog * prog
```

$$\frac{}{n_1 \ \text{+}\ n_2 \longrightarrow n_1 + n_2}$$

$$\frac{p_1 \longrightarrow p_1'}{p_1 \ \text{+}\ p_2 \longrightarrow p_1' \ \text{+}\ p_2} \qquad\qquad \frac{p_2 \longrightarrow p_2'}{p_1 \ \text{+}\ p_2 \longrightarrow p_1 \ \text{+}\ p_2'}$$

$$\frac{n_1 < n_2}{n_1 \ \text{<}\ n_2 \longrightarrow \text{true}} \qquad\qquad \frac{n_1 \geqslant n_2}{n_1 \ \text{<}\ n_2 \longrightarrow \text{false}}$$

$$\frac{p_1 \longrightarrow p_1'}{p_1 \ \text{<}\ p_2 \longrightarrow p_1' \ \text{<}\ p_2} \qquad\qquad \frac{p_2 \longrightarrow p_2'}{p_1 \ \text{<}\ p_2 \longrightarrow p_1 \ \text{<}\ p_2'}$$

$$\frac{}{\text{if true then } p_1 \text{ else } p_2 \longrightarrow p_1} \qquad \frac{}{\text{if false then } p_1 \text{ else } p_2 \longrightarrow p_2}$$

$$\frac{p \longrightarrow p'}{\text{if } p \text{ then } p_1 \text{ else } p_2 \longrightarrow \text{if } p' \text{ then } p_1 \text{ else } p_2}$$

Figure 1.1: Reduction rules.

A typical program would thus be

```
if 3 < 2 then 5 else 1                                    (1.5)
```

which would be encoded as the term

```
If (Lt (Int 3 , Int 2) , Int 5 , Int 1)
```

Reduction. Given programs p and p', we write $p \longrightarrow p'$ when p *reduces* to p'. This reduction relation is defined as the smallest one such that, for each of the rules listed in figure 1.1, if the relation above the horizontal bar holds, the relation below it also holds. In these rules, we write n_i for an arbitrary integer and the first rule indicates that the formal sum of two integers reduces to their sum (e.g. $3 \ \text{+}\ 2 \longrightarrow 5$).

An implementation of the reduction in OCaml is given in figure 1.2: given a program p, the function red either returns Some p' if there exists a program p' with $p \longrightarrow p'$ or None otherwise. Note that the reduction is not deterministic, i.e. a program can reduce to distinct programs:

$$5 \ \text{+}\ (5 \ \text{+}\ 4) \longleftarrow (3 \ \text{+}\ 2) \ \text{+}\ (5 \ \text{+}\ 4) \longrightarrow (3 \ \text{+}\ 2) \ \text{+}\ 9$$

The implementation provided in figure 1.2 chooses a particular reduction when there are multiple possibilities: we say that it implements a *reduction strategy*.

A program is *irreducible* when it does not reduce to another program. It can be remarked that

– values are irreducible,

– there are irreducible programs which are not values, e.g. 3 + true.

```
(** Perform one reduction step. *)
let rec red : prog -> prog option = function
  | Bool _ | Int _ -> None
  | Add (Int n1 , Int n2) -> Some (Int (n1 + n2))
  | Add (p1 , p2) ->
    (
      match red p1 with
      | Some p1' -> Some (Add (p1' , p2))
      | None ->
        match red p2 with
        | Some p2' -> Some (Add (p1 , p2'))
        | None     -> None
    )
  | Lt (Int n1 , Int n2) -> Some (Bool (n1 < n2))
  | Lt (p1 , p2) ->
    (
      match red p1 with
      | Some p1' -> Some (Lt (p1' , p2))
      | None ->
        match red p2 with
        | Some p2' -> Some (Lt (p1 , p2'))
        | None     -> None
    )
  | If (Bool true  , p1 , p2) -> Some p1
  | If (Bool false , p1 , p2) -> Some p2
  | If (p , p1 , p2) ->
    match red p with
    | Some p' -> Some (If (p' , p1 , p2))
    | None    -> None
```

Figure 1.2: Implementation of reduction.

$$\frac{}{\vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{bool}} \qquad \frac{n \in \mathbb{N}}{\vdash n : \texttt{int}}$$

$$\frac{\vdash p_1 : \texttt{int} \qquad \vdash p_2 : \texttt{int}}{\vdash p_1 \texttt{ + } p_2 : \texttt{int}} \qquad \frac{\vdash p_1 : \texttt{int} \qquad \vdash p_2 : \texttt{int}}{\vdash p_1 \texttt{ < } p_2 : \texttt{bool}}$$

$$\frac{\vdash p : \texttt{bool} \qquad \vdash p_1 : A \qquad \vdash p_2 : A}{\vdash \texttt{if } p \texttt{ then } p_1 \texttt{ else } p_2 : A}$$

Figure 1.3: Typing rules.

In the second case, the reason why the program `3 + true` cannot be further reduced is that an unexpected value was provided to the sum: we were hoping for an integer instead of the value `true`. We will see that the typing system precisely prevents such situations from arising.

Typing. A type in our language is either an integer (`int`) or a boolean (`bool`), which can be represented by the type

```
type t = TInt | TBool
```

We write $\vdash p : A$ to indicate that the program p has the type A and call it a *typing judgment.* This relation is defined inductively by the rules of figure 1.3. This means that a program p has type A when $\vdash p : A$ can be derived using the above rules. For instance, the program (1.5) has type `int`:

$$\frac{\dfrac{\vdash 3 : \texttt{int} \qquad \vdash 2 : \texttt{int}}{\vdash 3 \texttt{ < } 2 : \texttt{bool}} \qquad \frac{}{\vdash 5 : \texttt{int}} \qquad \frac{}{\vdash 1 : \texttt{int}}}{\vdash \texttt{if } 3 \texttt{ < } 2 \texttt{ then } 5 \texttt{ else } 1 : \texttt{int}}$$

Such a tree showing that a typing judgment is derivable is called a *derivation tree.* The principle of type checking and type inference algorithms of OCaml is to try to construct such a derivation of a typing judgment, using the above rules. In our small toy language, this is quite easy and is presented in figure 1.4. For a language with much more features as OCaml (where we have functions and polymorphism, not to mention objects or generalized algebraic data types) this is much more subtle, but still follows the same general principle.

It can be observed that a term can have at most one type. We can thus speak of *the* type of a typable program:

Theorem 1.4.3.1 (Uniqueness of types). Given a program p, if p is both of types A and A' then $A = A'$.

Proof. By induction on p: depending on the form of p, at most one rule applies. For instance, if p is of the form `if` p_0 `then` p_1 `else` p_2, the only rule which allows typing p is

$$\frac{\vdash p_0 : \texttt{bool} \qquad \vdash p_1 : A \qquad \vdash p_2 : A}{\vdash \texttt{if } p_0 \texttt{ then } p_1 \texttt{ else } p_2 : A}$$

```
exception Type_error

(** Infer the type of a program. *)
let rec infer = function
  | Bool _  -> TBool
  | Int _   -> TInt
  | Add (p1 , p2) ->
    check p1 TInt;
    check p2 TInt;
    TInt
  | Lt  (p1 , p2) ->
    check p1 TInt;
    check p2 TInt;
    TBool
  | If (p , p1 , p2) ->
    check p TBool;
    let t = infer p1 in
    check p2 t;
    t

(** Check that a program has a given type. *)
and check p t =
  if infer p <> t then raise Type_error
```

Figure 1.4: Type inference and type checking.

Since p_1 and p_2 admit at most one type A by induction hypothesis, p also does. Other cases are similar. $\square$

As explained in section 1.4.2, full-fledged languages such as OCaml do not generally satisfy such a strong property. The type of a program is not generally unique, but in good typing systems there exists instead a type which is "the most general".

Safety. We are now ready to formally state the safety properties guaranteed for typed programs. The first one, called *subject reduction*, states that the reduction preserves typing:

Theorem 1.4.3.2 (Subject reduction). Given programs p and p' such that $p \longrightarrow p'$, if p has type A then p' also has type A.

Proof. By hypothesis, we have both a derivation of $p \longrightarrow p'$ and $\vdash p : A$. We reason by induction on the former. For instance, suppose that the last rule is

$$\frac{p_1 \longrightarrow p'_1}{p_1 \ + \ p_2 \longrightarrow p'_1 \ + \ p_2}$$

The derivation of $\vdash p : A$ necessarily ends with

$$\frac{\vdash p_1 : \texttt{int} \qquad \vdash p_2 : \texttt{int}}{\vdash p_1 \ + \ p_2 : \texttt{int}}$$

In particular, we have $\vdash p_1 : \texttt{int}$ and thus, by induction hypothesis, $\vdash p'_1 : \texttt{int}$ is derivable. We conclude using the derivation

$$\frac{\vdash p'_1 : \texttt{int} \qquad \vdash p_2 : \texttt{int}}{\vdash p'_1 \ + \ p_2 : \texttt{int}}$$

Other cases are similar. $\square$

The second important property is called *progress*, and states that the program either is a value or reduces.

Theorem 1.4.3.3 (Progress). Given a program p of type A, either p is a value or there exists a program p' such that $p \longrightarrow p'$.

Proof. By induction on the derivation of $\vdash p : A$. For instance, suppose that the last rule is

$$\frac{\vdash p_1 : \texttt{int} \qquad \vdash p_2 : \texttt{int}}{\vdash p_1 \ + \ p_2 : \texttt{int}}$$

By induction hypothesis, the following cases can happen:

– $p_1 \longrightarrow p'_1$: in this case, we have $p_1 \ + \ p_2 \longrightarrow p'_1 \ + \ p_2$,

– $p_2 \longrightarrow p'_2$: in this case, we have $p_1 \ + \ p_2 \longrightarrow p_1 \ + \ p'_2$,

– p_1 and p_2 are values: in this case, they are necessarily integers and $p_1 \ + \ p_2$ reduces to their sum.

Other cases are similar. $\square$

The *safety* property finally states that typable programs never encounter errors, in the sense that their execution is never stuck: for instance, we will never try to evaluate a program such as `3 + true` during the reduction.

Theorem 1.4.3.4 (Safety)*.* A program p of type A is *safe*: either

  – p reduces to a value v in finitely many steps

$$p \longrightarrow p_1 \longrightarrow p_2 \longrightarrow \cdots \longrightarrow p_n \longrightarrow v$$

  – or p loops: there is an infinite sequence of reductions

$$p \longrightarrow p_1 \longrightarrow p_2 \longrightarrow \cdots$$

Proof. Consider a maximal sequence of reductions from p. If this sequence is finite, by maximality, its last element p' is an irreducible program. Since p is of type A and reduces to p', by the subject reduction theorem 1.4.3.2 p' also has type A. We can thus apply the progress theorem 1.4.3.3 and deduce that either p' is a value or there exists p'' such that $p' \longrightarrow p''$. The second case is impossible since it would contradict the maximality of the sequence of reductions. □

Of course, in our small language, a program cannot give rise to an infinite sequence of reductions, but the formulation and proof of the previous theorem will generalize to languages in which this is not the case. The previous properties of subject reduction and progress are entirely formalized in section 7.1.

Limitations of typing. The typing systems (such as the one described above or the one of OCaml) reject legit programs such as

```
(if true then 3 else false) + 1
```

which reduces to a value. Namely, the system imposes the requirement that the two branches of a conditional branching should have the same type, which is not the case here, even though we know that only the first branch will be taken, because the condition is the constant boolean `true`. We thus ensure that typable programs are safe, but not that all safe programs are typable. In fact, this has to be this way since an easy reduction to the halting problem shows that the safety of programs is undecidable as soon as the language is rich enough.

Also, the typing system does not prevent all errors from occurring during the execution, such as dividing by zero or accessing an array out of its bounds. This is because the typing system is not expressive enough. For instance, the function

```
let f x = 1 / (x - 2)
```

should intuitively be given the type

$$\{n : \mathtt{int} \mid n \neq 2\} \to \mathtt{int}$$

which states this function is correct as long as its input is an integer different from 2, but this is of course not a valid type in OCaml. We will see in chapters 6 and 8 that some languages do allow such rich typing, at the cost of losing type inference (but type checking is still decidable).

1.5 Typing as proving

We would now like to give the intuition for the main idea of this course, that programs correspond to proofs. Understanding in details this correspondence will allow us to design very rich typing systems which allow to formally prove fine theorems and reason about programs.

1.5.1 Arrow as implication. As a first illustration of this, we will see here that simple types (such as the ones used in OCaml) can be read as propositional formulas. The translation is simply a matter of slightly changing the way we read types: a type variable `'a` can be read as a propositional variable A and the arrow `->` can be read as an implication $\Rightarrow$. Now, we can observe that there is a program of a given type (in a reasonable subset of OCaml, see below) precisely when the corresponding formula is true (for a reasonable notion of true formula). For instance, we expect that the formula

$$A \Rightarrow A \qquad \text{corresponding to the type} \qquad \text{'a -> 'a}$$

is provable. And indeed, there is a program of this type, the identity:

```
let id : 'a -> 'a = fun x -> x
```

We have specified here the type of this function for clarity. We can give many other such examples. For instance, $A \Rightarrow B \Rightarrow A$ is proved by

```
let k : 'a -> 'b -> 'a = fun x y -> x
```

The formula $(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$ can be proved by the composition

```
let comp : ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c) =
  fun f g x -> g (f x)
```

The formula $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$ is proved by

```
let s : ('a -> 'b -> 'c) -> ('a -> 'b) -> ('a -> 'c) =
  fun f g x -> f x (g x)
```

and so on.

Remark 1.5.1.1. In general, there is not a unique proof of a given formula. For instance, $A \Rightarrow A$ can also be proved by

```
fun x -> k x 3
```

where `k` is the function defined above.

1.5.2 Other connectives. For now, the fragment of the logic we have is very poor (we only have implication as connective), but other usual connectives also have counterparts in types.

Conjunction. A *conjunction* proposition $A \wedge B$ means that both A and B hold. In terms of types, the counterpart is a product:

$$A \wedge B \qquad \text{corresponds to} \qquad \text{'a * 'b}$$

and we have programs implementing usual propositions such as $A \wedge B \Rightarrow A$:

```
let proj1 : ('a * 'b) -> 'a = fun (a , b) -> a
```

or the commutativity of conjunction $A \wedge B \Rightarrow B \wedge A$:

```
let comm : ('a * 'b) -> ('b * 'a) = fun (a , b) -> b , a
```

Truth. The formula $\top$ corresponding to *truth* is always provable and we expect that there is exactly one reason for which it should be true. Thus

$$\top \qquad \text{corresponds to} \qquad \text{unit}$$

and we can prove $A \Rightarrow \top$:

```
let unit_intro : 'a -> unit = fun x -> ()
```

Falsity. The formula $\bot$ corresponds to *falsity* and we do not expect that it can be proved (because false is never true). We can make it correspond to the `empty` type, which can be defined as a type with no constructor:

```
type empty = |
```

The formula $\bot \Rightarrow A$ is then shown by

```
let empty_elim : empty -> 'a = fun x -> match x with _ -> .
```

(the "." is a "refutation case" meaning that the compiler should ensure that this case should never happen, it is almost never used in OCaml unless you are doing tricky stuff such as the above).

Negation. The *negation* can then be defined as usual by $\neg A$ being a notation for $A \Rightarrow \bot$, and we can prove the reasoning by contraposition

$$(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$$

by

```
let contr : ('a -> 'b) -> (('b -> empty) -> ('a -> empty)) =
  fun f g a -> g (f a)
```

or $A \Rightarrow \neg\neg A$ by

```
let nni : 'a -> (('a -> empty) -> empty) = fun a f -> f a
```

Disjunction. A *disjunction* formula $A \lor B$ can be thought of as being either A or B. We can implement it as a *coproduct* type, which is an inductive type where a value is either a value of type `'a` or a value of type `'b`, see section 1.3.2:

```
type ('a , 'b) coprod = Left of 'a | Right of 'b
```

We can then prove the formula $A \lor B \Rightarrow B \lor A$, stating that disjunction is commutative, by

```
let comm : ('a , 'b) coprod -> ('b , 'a) coprod = fun x ->
  match x with
  | Left  a -> Right a
  | Right b -> Left  b
```

or the distributivity $A \land (B \lor C) \Rightarrow (A \land B) \lor (A \land C)$ of conjunction over disjunction by

```
let dist : ('a * ('b , 'c) coprod) -> ('a * 'b , 'a * 'c) coprod =
  fun (a , x) ->
  match x with
  | Left  b -> Left  (a , b)
  | Right c -> Right (a , c)
```

or the de Morgan formula $(\lnot A \lor B) \Rightarrow (A \Rightarrow B)$ by

```
let de_Morgan : ('a -> empty, 'b) coprod -> ('a -> 'b) = fun x a ->
  match x with
  | Left  f -> empty_elim (f a)
  | Right b -> b
```

1.5.3 Limitations of the correspondence. This correspondence has some limitations due to the fact that OCaml is, after all, a language designed to do programming, not logic. It is easy to prove formulas which are not true if we use "advanced" features of the language such as exceptions. For instance, the following "proves" $A \Rightarrow B$:

```
let absurd : 'a -> 'b = fun x -> raise Not_found
```

More annoying counter-examples come from functions which are not terminating (i.e. looping). For instance, we can also "prove" $A \Rightarrow B$ by

```
let rec absurd : 'a -> 'b = fun x -> absurd x
```

Note that, in particular, both allow to "prove" $\bot$:

```
let fake : empty = absurd ()
```

Finally, we can notice that there does not seem to be any reasonable way to implement the classical formula $\lnot A \lor A$ (apart from using the above tricks), which would correspond to a program of the type

```
('a -> empty , 'a) coprod
```

In next chapters, we will see that it is indeed possible to design languages in which a formula is provable precisely when there is a program of the corresponding type. Such languages do not have functions with "side-effects" (such as raising an exception) and enforce that all the programs are terminating.

Propositional logic

In this chapter, we present propositional logic: this is the fragment of logic consisting of propositions (very roughly, something which can either be true or false) joined by connectives. We will see various ways of formalizing the proofs in propositional logic – with a particular focus on natural deduction – and study the properties of those. We begin with the formalism of natural deduction in section 2.2, show that it enjoys the cut elimination property in section 2.3 and discuss strategies for searching for proofs in section 2.4. The classical variant of logic is presented in section 2.5. We then present two alternative logical formalisms: sequent calculus (section 2.6) and Hilbert calculus (section 2.7). Finally, we introduce Kripke semantics in section 2.8, which can be considered as an intuitionistic counterpart of boolean models for classical logic.

2.1 Introduction

2.1.1 From provability to proofs. Most of you are acquainted with boolean logic based on the booleans, which we write here as 0 for false, and 1 for true. In this setting, every propositional formula can be interpreted as a boolean, provided that we have an interpretation for the variables. The truth tables for usual connectives are

$A \wedge B$	0	1
0	0	0
1	0	1

$A \vee B$	0	1
0	0	1
1	1	1

$A \Rightarrow B$	0	1
0	1	1
1	0	1

For instance, we know that the formula $A \Rightarrow A$ is valid because, for whichever interpretation of A as a boolean, the induced interpretation of the formula is 1.

We have this idea that propositions should correspond to types. Therefore, rather than booleans, propositions should be interpreted as sets of values and implications as functions between the corresponding values. For instance, if we write $\mathbb{N}$ for a proposition interpreted as the set of natural numbers, the type $\mathbb{N} \Rightarrow \mathbb{N}$ would correspond to the set of functions from natural numbers to themselves. We now see that the boolean interpretation is very weak: it only cares about whether sets are empty or not. For instance, depending on whether X is empty ($\emptyset$) or non-empty ($\neg\emptyset$), the following table indicates whether the set $X \to X$ of functions from X to X is empty or not:

$A \to B$	$\emptyset$	$\neg\emptyset$
$\emptyset$	$\neg\emptyset$	$\neg\emptyset$
$\neg\emptyset$	$\emptyset$	$\neg\emptyset$

Reading $\emptyset$ as "false" and $\neg\emptyset$ as "true", we see that we recover the usual truth table for implication. In this sense, the fact that the formula $\mathbb{N} \Rightarrow \mathbb{N}$ is true only shows that there exists such a function, but in fact there are many such functions, and we would be able to reason about the various functions themselves.

Actually, the interpretation of implications as sets of functions is still not entirely satisfactory because, given a function of type $\mathbb{N} \to \mathbb{N}$, there are many ways to implement it. We could have programs of different complexities, using their arguments in different ways, and so on. For instance, the constant function $x \mapsto 0$ can be implemented as

```
let f x = 0
```

or

```
let f x = x - x
```

or

```
let rec f x = if x = 0 then x else f (x - 1)
```

respectively in constant, logarithmic and linear complexity (if we assume the predecessor to be computed in constant time). We thus want to shift from an *extensional* perspective, where two functions are equal when they have the same values on the same inputs, to an *intentional* one where the way the result is computed matters. This means that we should be serious about what is a program, or equivalently a proof, and define it precisely so that we can reason about the proofs of a proposition instead of its provability: we want to know what the proofs are and not only whether there exist one or not. This is the reason why Girard advocates that there are three levels for interpreting proofs [Gir11, section 7.1]:

0. the *boolean level*: propositions are interpreted as booleans and we are interested in whether a proposition is provable or not,

1. the *extensional level*: propositions are interpreted as sets and we are interested in which functions can be implemented,

2. the *intentional level*: we are interested in the proofs themselves (and how they evolve via cut elimination).

2.1.2 Intuitionism. This shift from provability to proofs was started by the philosophical position of Brouwer starting in the early twentieth century, called *intuitionism*. According to this point of view, mathematics does not consist in discovering the properties of a preexisting objective reality, but is rather a mental subjective construction, which is independent of the reality and has an existence on its own, whose validity follows from the intuition of the mathematician. From this point of view

– the conjunction $A \wedge B$ of two propositions should be seen as having both a proof of A and a proof of B: if we interpret propositions as sets, $A \wedge B$ should not be interpreted as the intersection $A \cap B$, but rather are the product $A \times B$,

– a disjunction $A \vee B$ should be interpreted as having a proof of A or a proof of B, i.e. it does not correspond to the union $A \cup B$, but rather to the disjoint union $A \sqcup B$,

– an implication $A \Rightarrow B$ should be interpreted as having a way to construct a proof of B from a proof of A,

– a negation $\neg A = A \Rightarrow \bot$ should be interpreted as having a counter-example to A, i.e. a way to produce an absurdity from a proof of A.

Interestingly, this led Brouwer to reject principles which are classically valid. For instance, according to this point of view $\neg\neg A$ should not be considered as equivalent to A because the implication

$$\neg\neg A \Rightarrow A$$

should not hold: if we can show that there is no counter-example to A, this does not mean that we actually have a proof of A. For instance, suppose that I cannot find my key inside my apartment and my door is locked: I must have locked my door so that I know that my key is somewhere in the apartment and it is not lost, but I still cannot find it. Not having lost my key (i.e. not not having my key) does not mean that I have my key; in other words,

$$\neg\neg\mathrm{Key} \Rightarrow \mathrm{Key}$$

does not hold (explanation borrowed from Ingo Blechschmidt). For similar reasons, Brouwer also rejected the excluded middle

$$\neg A \vee A$$

given an arbitrary proposition A: in order to have a proof for it, we should have a way, whichever the proposition A is, to produce a counter-example to it or a proof of it. Logic rejecting these principles is called *intuitionistic* and, by opposition, we speak of *classical* logic when they are admitted.

2.1.3 Formalizing proofs. Our goal is to give a precise definition of what a proof is. This will be done by formalizing the rules using which we usually construct our reasoning. For instance, suppose that we want to prove that the function $x \mapsto 2 \times x$ is continuous in 0: we have to prove the formula

$$\forall\varepsilon.(\varepsilon > 0 \Rightarrow \exists\eta.(\eta > 0 \wedge \forall x.|x| < \eta \Rightarrow |2x| < \varepsilon))$$

This is done in the following steps, resulting in the following transformed formulas to be proved.

– Suppose given ε, we have to show:

$$\varepsilon > 0 \Rightarrow \exists\eta.(\eta > 0 \wedge \forall x.|x| < \eta \Rightarrow |2x| < \varepsilon)$$

– Suppose that $\varepsilon > 0$ holds, we have to show:

$$\exists\eta.(\eta > 0 \wedge \forall x.|x| < \eta \Rightarrow |2x| < \varepsilon)$$

– Take $\eta = \varepsilon/2$, we have to show:

$$\varepsilon/2 > 0 \wedge \forall x.|x| < \varepsilon/2 \Rightarrow |2x| < \varepsilon$$

– We have to show both

$$\varepsilon/2 > 0 \qquad \text{and} \qquad \forall x.|x| < \varepsilon/2 \Rightarrow |2x| < \varepsilon$$

- For $\varepsilon/2 > 0$:

    - because $2 > 0$, this amounts to showing $(\varepsilon/2) \times 2 > 0 \times 2$,

    - which, by usual identities, amounts to showing $\varepsilon > 0$,

    - which is an hypothesis.

- For $\forall x.|x| < \varepsilon/2 \Rightarrow |2x| < \varepsilon$:

    - suppose given x, we have to show: $|x| < \varepsilon/2 \Rightarrow |2x| < \varepsilon$,

    - suppose that $|x| < \varepsilon/2$ holds, we have to show: $|2x| < \varepsilon$,

    - since $2 > 0$, this amounts to showing: $|2x|/2 < \varepsilon/2$,

    - which, by usual identities, amounts to showing: $|x| < \varepsilon/2$,

    - which is an hypothesis.

Now that we have decomposed the proof into very small steps, it seems possible to give a list of all the generic rules that we are allowed to apply in a reasoning. We will do so and will introduce a convenient formalism and notations, so that the above proof will be written as:

$$
\cfrac{
\cfrac{\cfrac{\cfrac{}{\varepsilon > 0 \vdash \varepsilon > 0}}{\varepsilon > 0 \vdash (\varepsilon/2) \times 2 > 0 \times 2}}{\varepsilon > 0 \vdash \varepsilon/2 > 0}
\qquad
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\varepsilon > 0, |x| < \varepsilon/2 \vdash |x| < \varepsilon/2}}{\varepsilon > 0, |x| < \varepsilon/2 \vdash |2x|/2 < \varepsilon/2}}{\varepsilon > 0, |x| < \varepsilon/2 \vdash |2x| < \varepsilon}}{\varepsilon > 0 \vdash |x| < \varepsilon/2 \Rightarrow |2x| < \varepsilon}}{\varepsilon > 0 \vdash \forall x.|x| < \varepsilon/2 \Rightarrow |2x| < \varepsilon}
}{
\cfrac{\cfrac{\cfrac{\varepsilon > 0 \vdash \varepsilon/2 > 0 \wedge \forall x.|x| < \varepsilon/2 \Rightarrow |2x| < \varepsilon}{\varepsilon > 0 \vdash \exists \eta.(\eta > 0 \wedge \forall x.|x| < \eta \Rightarrow |2x| < \varepsilon)}}{\vdash \varepsilon > 0 \Rightarrow \exists \eta.(\eta > 0 \wedge \forall x.|x| < \eta \Rightarrow |2x| < \varepsilon)}}{\vdash \forall \varepsilon.(\varepsilon > 0 \Rightarrow \exists \eta.(\eta > 0 \wedge \forall x.|x| < \eta \Rightarrow |2x| < \varepsilon))}
}
$$

(when read from bottom to top, you should be able to see the precise correspondence with the previous description of the proof).

2.1.4 Properties of the logical system. Once we have formalized our logical system we should do some sanity checks. The first requirement is that it should be *consistent*: there is at least one formula A which is not provable (otherwise, the system would be entirely pointless). The second requirement is that typechecking should be decidable: there should be an algorithm which checks whether a proof is valid or not. In contrast, the question of whether a formula is provable or not will not be decidable in general and we do not expect to have an algorithm for that.

2.2 Natural deduction

Natural deduction is the first formalism for proofs that we will study. It was introduced by Gentzen [Gen35]. We first present the intuitionistic version.

2.2.1 Formulas. We suppose fixed a countably infinite set $\mathcal{X}$ of *propositional variables*. The set $\mathcal{A}$ of *formulas* or *propositions* is generated by the following grammar

$$A, B ::= X \mid A \Rightarrow B \mid A \wedge B \mid \top \mid A \vee B \mid \bot \mid \neg A$$

where X is a propositional variable (in $\mathcal{X}$) and A and B are propositions. They are respectively read as a propositional variable, *implication*, *conjunction*, *truth*, *disjunction*, *falsity* and *negation*. By convention, $\neg$ binds the most tightly, then $\wedge$, then $\vee$, then $\Rightarrow$:

$$\neg A \vee B \wedge C \Rightarrow D \qquad \text{reads as} \qquad ((\neg A) \vee (B \wedge C)) \Rightarrow D$$

Moreover, all binary connectives are implicitly bracketed to the right:

$$A_1 \wedge A_2 \wedge A_3 \Rightarrow B \Rightarrow C \qquad \text{reads as} \qquad (A_1 \wedge (A_2 \wedge A_3)) \Rightarrow (B \Rightarrow C)$$

This is particularly important for $\Rightarrow$, for the connectives $\wedge$ and $\vee$ the other convention could be chosen with almost no impact. We sometimes write $A \Leftrightarrow B$ for $(A \Rightarrow B) \wedge (B \Rightarrow A)$.

A *subformula* of a formula A is a formula occurring in A. The set of subformulas of A can formally be defined by induction on A by

$$\operatorname{Sub}(X) = \{X\} \qquad \operatorname{Sub}(A \Rightarrow B) = \{A \Rightarrow B\} \cup \operatorname{Sub}(A) \cup \operatorname{Sub}(B)$$
$$\operatorname{Sub}(\top) = \{\top\} \qquad \operatorname{Sub}(A \wedge B) = \{A \wedge B\} \cup \operatorname{Sub}(A) \cup \operatorname{Sub}(B)$$
$$\operatorname{Sub}(\bot) = \{\bot\} \qquad \operatorname{Sub}(A \vee B) = \{A \vee B\} \cup \operatorname{Sub}(A) \cup \operatorname{Sub}(B)$$
$$\operatorname{Sub}(\neg A) = \{\neg A\} \cup \operatorname{Sub}(A)$$

2.2.2 Sequents. A *context*

$$\Gamma = A_1, \ldots, A_n$$

is a list of propositions. A *sequent*, or *judgment*, is a pair

$$\Gamma \vdash A$$

consisting of a context Γ and a variable A. Such a sequent should be read as "under the hypothesis in Γ, I can prove A" or "supposing that I can prove the propositions in Γ, I can prove A". The comma in a context can thus be read as a "meta" conjunction (the logical conjunction being $\wedge$) and the sign $\vdash$ as a "meta" implication (the logical implication being $\Rightarrow$).

Remark 2.2.2.1. The notation derives from Frege's *Begriffsschrift* [Fre79], an axiomatization of first-order logic based on a graphical notation, in which logical connectives are drawn by using wires of particular shapes: the formulas $\neg A$, $A \Rightarrow B$ and $\forall x.A$ are respectively drawn as

$$\dashv A \qquad\qquad \begin{array}{l} A \\ B \end{array} \qquad\qquad \neg \!\!\multimap\!\!\!\!{\scriptstyle x}\!\!\multimap\!\! A$$

In this system, given a proposition drawn $\longrightarrow A$, the notation $\vdash\!\!\longrightarrow A$ means that A is provable. The assertion that $(\forall x.A) \Rightarrow (\exists x.B)$ is provable would for instance be written

$$\vdash\!\!\!\begin{array}{l} \multimap x \multimap A \\ \multimap x \multimap B \end{array}$$

(in classical logic, the formula $\exists x.B$ is equivalent to $\neg \forall x.\neg B$). The symbol $\vdash$ used in sequents, as well as the symbol $\neg$ for negation, originate from there.

$$\frac{}{\Gamma, A, \Gamma' \vdash A}(\text{ax})$$

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}(\Rightarrow_{\text{E}}) \qquad\qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}(\Rightarrow_{\text{I}})$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}(\wedge_{\text{E}}^{\text{l}}) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}(\wedge_{\text{E}}^{\text{r}}) \qquad\qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}(\wedge_{\text{I}})$$

$$\frac{}{\Gamma \vdash \top}(\top_{\text{I}})$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}(\vee_{\text{E}}) \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}(\vee_{\text{I}}^{\text{l}}) \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}(\vee_{\text{I}}^{\text{r}})$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}(\bot_{\text{E}})$$

$$\frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \bot}(\neg_{\text{E}}) \qquad\qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}(\neg_{\text{I}})$$

Figure 2.1: NJ: rules of intuitionistic natural deduction.

2.2.3 Inference rules. An *inference rule*, written

$$\frac{\Gamma_1 \vdash A_1 \qquad \ldots \qquad \Gamma_n \vdash A_n}{\Gamma \vdash A} \tag{2.1}$$

consists of n sequents $\Gamma_i \vdash A_i$, called the *premises* of the rule, and a sequent $\Gamma \vdash A$, called the *conclusion* of the rule. We sometimes identify the rules by a name given to them, which is written on the right of the rule. Some rules also come with external hypothesis on the formulas occurring in the premises: those are called *side conditions*. There are two ways to read an inference rule:

- the *deductive* way, from top to bottom: from a proof for each of the premises $\Gamma_i \vdash A_i$ we can deduce $\Gamma \vdash A$,

- the *inductive* or *proof search* way, from bottom to top: if we want to prove $\Gamma \vdash A$ by that inference rule we need to prove all the premises $\Gamma_i \vdash A_i$.

Both are valid ways of thinking about proofs, but one might be more natural than the other one depending on the application.

2.2.4 Intuitionistic natural deduction. The rules for *intuitionistic natural deductions* are shown in figure 2.1, the resulting system often being called NJ (N for natural deduction and J for intuitionistic). Apart from the *axiom* rule (ax), each rule is specific to a connective and the rules can be classified in two families depending on whether this connective appears in the conclusion or in the premises:

    – the *elimination rules* allow the use of a formula with a given connective (which is in the formula in the leftmost premise, called the *principal premise*),

    – the *introduction rules* construct a formula with a given connective.

In figure 2.1, the elimination (resp. introduction) rules are displayed on the left (resp. right) and bear names of the form $(\ldots_\mathrm{E})$ (resp. $(\ldots_\mathrm{I})$).

    The axiom rule allows the use of a formula in the context Γ: supposing that a formula A holds, we can certainly prove it. This rule is the only one to really make use of the context: when read from the bottom to top, all the other rules either propagate the context or add hypothesis to it, but never inspect it.

    The introduction rules are the most easy to understand: they allow proving a formula with a given logical connective from the proofs of the immediate subformulas. For instance, $(\wedge_\mathrm{I})$ states that from a proof of A and a proof of B, we can construct a proof of $A \wedge B$. Similarly, the rule $(\Rightarrow_\mathrm{I})$ follows the usual reasoning principle for implication: if, after supposing that A holds, we can show B, then $A \Rightarrow B$ holds.

    In contrast, the elimination rules allow the use of a connective. For instance, the rule $(\Rightarrow_\mathrm{E})$, which is traditionally called *modus ponens* or *detachment rule*, says that if A implies B and A holds then certainly B must hold. The rule $(\vee_\mathrm{E})$ is more subtle and corresponds to a case analysis: if we can prove $A \vee B$ then, intuitively, we can prove A or we can prove B. If in both cases we can deduce C then C must hold. The elimination rule $(\bot_\mathrm{E})$ is sometimes called *ex falso quodlibet* or the *explosion principle*: it states that if we can prove false then the whole logic collapses, and we can prove anything.

    We can notice that there is no elimination rule for $\top$ (knowing that $\top$ is true does not bring any new information), and no introduction rule for $\bot$ (we do not expect that there is a way to prove falsity). There are two elimination rules for $\wedge$ which are respectively called *left* and *right* rules, and similarly there are two introduction rules for $\vee$.

2.2.5 Proofs. The set of *proofs* (or *derivations*) is the smallest set such that given proofs π_i of the sequent $\Gamma_i \vdash A_i$, for $1 \leqslant i \leqslant n$, and an inference rule of the form (2.1), there is a proof of $\Gamma \vdash A$, often written in the form of a tree as

$$\dfrac{\dfrac{\pi_1}{\Gamma_1 \vdash A_1} \quad \cdots \quad \dfrac{\pi_n}{\Gamma_n \vdash A_n}}{\Gamma \vdash A}$$

A sequent $\Gamma \vdash A$ is *provable* (or *derivable*) when it is the conclusion of a proof. A formula A is *provable* when it is provable without hypothesis, i.e. when the sequent $\vdash A$ is provable.

Example 2.2.5.1. The formula $(A \wedge B) \Rightarrow (A \vee B)$ is provable (for any formulas A and B):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B} \ (\mathrm{ax})}{A \wedge B \vdash A} \ (\wedge_\mathrm{E})}{A \wedge B \vdash A \vee B} \ (\vee_\mathrm{I}^\mathrm{l})}{\vdash A \wedge B \Rightarrow A \vee B} \ (\Rightarrow_\mathrm{I})$$

Example 2.2.5.2. The formula $(A \vee B) \Rightarrow (B \vee A)$ is provable:

$$
\cfrac{
\cfrac{}{A \vee B \vdash A \vee B}\ (\text{ax}) \qquad
\cfrac{\cfrac{}{A \vee B, A \vdash A}\ (\text{ax})}{A \vee B, A \vdash B \vee A}\ (\vee_I^r) \qquad
\cfrac{\cfrac{}{A \vee B, B \vdash B}\ (\text{ax})}{A \vee B, B \vdash B \vee A}\ (\vee_I^l)
}{
\cfrac{A \vee B \vdash B \vee A}{\vdash A \vee B \Rightarrow B \vee A}\ (\Rightarrow_I)
}\ (\vee_E)
$$

Example 2.2.5.3. The formula $A \Rightarrow \neg\neg A$ is provable:

$$
\cfrac{
\cfrac{
\cfrac{\cfrac{}{A, \neg A \vdash \neg A}\ (\text{ax}) \qquad \cfrac{}{A, \neg A \vdash A}\ (\text{ax})}{A, \neg A \vdash \bot}\ (\neg_E)
}{A \vdash \neg\neg A}\ (\neg_I)
}{\vdash A \Rightarrow \neg\neg A}\ (\Rightarrow_I)
$$

Example 2.2.5.4. The formula $(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$ is provable:

$$
\cfrac{
\cfrac{
\cfrac{
\cfrac{}{A \Rightarrow B, \neg B, A \vdash \neg B}\ (\text{ax}) \qquad
\cfrac{\cfrac{}{A \Rightarrow B, \neg B, A \vdash A \Rightarrow B}\ (\text{ax}) \quad \cfrac{}{A \Rightarrow B, \neg B, A \vdash A}\ (\text{ax})}{A \Rightarrow B, \neg B, A \vdash B}\ (\Rightarrow_E)
}{A \Rightarrow B, \neg B, A \vdash \bot}\ (\neg_E)
}{
\cfrac{A \Rightarrow B, \neg B \vdash \neg A}{\cfrac{A \Rightarrow B \vdash \neg B \Rightarrow \neg A}{\vdash (A \Rightarrow B) \Rightarrow \neg B \Rightarrow \neg A}\ (\Rightarrow_I)}\ (\Rightarrow_I)
}\ (\neg_I)
}{}
$$

Example 2.2.5.5. The formula $(\neg A \vee B) \Rightarrow (A \Rightarrow B)$ is provable:

$$
\cfrac{
\cfrac{}{\neg A \vee B, A \vdash \neg A \vee B}\ (\text{ax}) \quad
\cfrac{
\cfrac{\cfrac{\cfrac{}{\neg A \vee B, A, \neg A \vdash \neg A}\ (\text{ax}) \ \cfrac{}{\neg A \vee B, A, \neg A \vdash A}\ (\text{ax})}{\neg A \vee B, A, \neg A \vdash \bot}\ (\neg_E)}{\neg A \vee B, A, \neg A \vdash B}\ (\bot_E) \quad \cfrac{}{\neg A \vee B, A, B \vdash B}\ (\text{ax})
}{\neg A \vee B, A \vdash B}\ (\vee_E)
}{
\cfrac{\neg A \vee B \vdash A \Rightarrow B}{\vdash (\neg A \vee B) \Rightarrow (A \Rightarrow B)}\ (\Rightarrow_I)
}\ (\Rightarrow_I)
$$

Other typical provable formulas are

- $\wedge$ and $\top$ satisfy the axioms of idempotent commutative monoids:

$$(A \wedge B) \wedge C \Leftrightarrow A \wedge (B \wedge C) \qquad\qquad A \wedge B \Leftrightarrow B \wedge A$$
$$\top \wedge A \Leftrightarrow A \Leftrightarrow A \wedge \top \qquad\qquad A \wedge A \Leftrightarrow A$$

- $\vee$ and $\bot$ satisfy the axioms of idempotent commutative monoids

- $\wedge$ distributes over $\vee$ and conversely:

$$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$$
$$A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$$

- $\Rightarrow$ is reflexive and transitive

$$A \Rightarrow A \qquad\qquad (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$$

– currying:
$$((A \wedge B) \Rightarrow C) \Leftrightarrow (A \Rightarrow (B \Rightarrow C))$$

– usual reasoning structures with latin names, such as

$$(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A) \qquad \text{(modus tollens)}$$
$$(A \vee B) \Rightarrow (\neg A \Rightarrow B) \qquad \text{(modus tollendo ponens)}$$
$$\neg(A \wedge B) \Rightarrow (A \Rightarrow \neg B) \qquad \text{(modus ponendo tollens)}$$

Reasoning on proofs. In this formalism, the proofs are defined inductively and therefore we can reason by induction on them, which is often useful. Precisely, the induction principle on proofs is the following one:

Theorem 2.2.5.6 (Induction on proofs). Suppose given a predicate $P(\pi)$ on proofs π. Suppose moreover that for every rule of figure 2.1 and every proof π ending with this rule

$$\pi \quad = \quad \cfrac{\cfrac{\pi_1}{\Gamma_1 \vdash A_1} \quad \cdots \quad \cfrac{\pi_n}{\Gamma_n \vdash A_n}}{\Gamma \vdash A}$$

if $P(\pi_i)$ holds for every index i, with $1 \leqslant i \leqslant n$, then $P(\pi)$ also holds. Then $P(\pi)$ holds for every proof π.

2.2.6 Fragments. A *fragment* of intuitionistic logic is a system obtained by restricting to formulas containing only certain connectives and the rules concerning these connectives. By convention, the axiom rule (ax) is present in every fragment. For instance, the *implicational fragment* of intuitionistic logic is obtained by restricting to implication: formulas are generated by the grammar

$$A, B ::= X \mid A \Rightarrow B$$

and the rules are

$$\cfrac{}{\Gamma, A, \Gamma' \vdash A} \text{ (ax)} \qquad \cfrac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} (\Rightarrow_{\text{E}}) \qquad \cfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} (\Rightarrow_{\text{I}})$$

The *cartesian fragment* is obtained by restricting to product and implication. Another useful fragment is *minimal logic* obtained by considering formulas without $\perp$, and thus removing the rule ($\perp_{\text{E}}$).

2.2.7 Admissible rules. A rule is *admissible* when, whenever the premises are provable, the conclusion is also provable. An important point here is that the way the proof of the conclusion is constructed might depend on the proofs of the premises, and not only on the fact that we know that the premises are provable.

Structural rules. We begin by showing that the *structural rules* are admissible. Those rules are named in this way because they concern the structure of the logical proofs, as opposed to the particular connectives we are considering for formulas. They express some resource management possibilities for the hypotheses in sequents: we can permute, merge and weaken them, see section 2.2.10.

A first admissible rule is the weakening rule, which states that whenever one can prove a formula with some hypotheses, we can still prove it with more hypotheses. The proof with more hypotheses is "weaker" in the sense that it apply in less cases (since more hypotheses have to be satisfied).

Proposition 2.2.7.1 (Weakening). The *weakening rule*

$$\frac{\Gamma, \Gamma' \vdash B}{\Gamma, A, \Gamma' \vdash B} \; (\text{wk})$$

is admissible.

Proof. By induction on the proof of the hypothesis $\Gamma, \Gamma' \vdash B$.

– If the proof is of the form

$$\frac{}{\Gamma, \Gamma' \vdash B} \; (\text{ax})$$

with B occurring in Γ or Γ', then we conclude with

$$\frac{}{\Gamma, A, \Gamma' \vdash B} \; (\text{ax})$$

– If the proof is of the form

$$\frac{\begin{array}{c}\pi_1 \\ \overline{\Gamma, \Gamma' \vdash B \Rightarrow C}\end{array} \quad \begin{array}{c}\pi_2 \\ \overline{\Gamma, \Gamma' \vdash B}\end{array}}{\Gamma, \Gamma' \vdash C} \; (\Rightarrow_{\text{E}})$$

then we conclude with

$$\frac{\begin{array}{c}\pi'_1 \\ \overline{\Gamma, A, \Gamma' \vdash B \Rightarrow C}\end{array} \quad \begin{array}{c}\pi'_2 \\ \overline{\Gamma, A, \Gamma' \vdash B}\end{array}}{\Gamma, A, \Gamma' \vdash C} \; (\Rightarrow_{\text{E}})$$

where π'_1 and π'_2 are respectively obtained from π_1 and π_2 by induction hypothesis:

$$\pi'_1 = \frac{\begin{array}{c}\pi_1 \\ \overline{\Gamma, \Gamma' \vdash B \Rightarrow C}\end{array}}{\Gamma, A, \Gamma' \vdash B \Rightarrow C} \; (\text{wk}) \qquad \pi'_2 = \frac{\begin{array}{c}\pi_2 \\ \overline{\Gamma, \Gamma' \vdash B}\end{array}}{\Gamma, A, \Gamma' \vdash B} \; (\text{wk})$$

– If the proof is of the from

$$\frac{\begin{array}{c}\pi \\ \overline{\Gamma, \Gamma', B \vdash C}\end{array}}{\Gamma, \Gamma' \vdash B \Rightarrow C} \; (\Rightarrow_{\text{I}})$$

then we conclude with

$$\frac{\begin{array}{c}\pi' \\ \overline{\Gamma, A, \Gamma', B \vdash C}\end{array}}{\Gamma, A, \Gamma' \vdash B \Rightarrow C} \; (\Rightarrow_{\text{I}})$$

where π' is obtained from π by induction hypothesis.

– Other cases are similar. □

Also admissible is the exchange rule, which states that we can reorder hypothesis in the contexts:

Proposition 2.2.7.2 (Exchange)*.* The *exchange rule*

$$\frac{\Gamma, A, B, \Gamma' \vdash C}{\Gamma, B, A, \Gamma' \vdash C} \ (\text{xch})$$

is admissible.

Proof. By induction on the proof of the hypothesis $\Gamma, A, B, \Gamma' \vdash C$. □

Given a proof π of some sequent, we often write $w(\pi)$ for a proof obtained by weakening. Another admissible rule is contraction, which states that if we can prove a formula with two occurrences of a hypothesis, we can also prove it with one occurrence.

Proposition 2.2.7.3 (Contraction)*.* The *contraction rule*

$$\frac{\Gamma, A, A, \Gamma' \vdash B}{\Gamma, A, \Gamma' \vdash B} \ (\text{contr})$$

is admissible.

Proof. By induction on the proof of the hypothesis $\Gamma, A, A, \Gamma' \vdash B$. □

We can also formalize the fact that knowing $\top$ does not bring information, what we call here *truth strengthening* (we are not aware of a standard terminology for this one):

Proposition 2.2.7.4 (Truth strengthening)*.* The following rule is admissible:

$$\frac{\Gamma, \top, \Gamma' \vdash A}{\Gamma, \Gamma' \vdash A} \ (\text{tstr})$$

Proof. By induction on the proof of the hypothesis $\Gamma, \top, \Gamma' \vdash A$, the only "subtle" case being that we have to transform

$$\frac{}{\Gamma, \top, \Gamma' \vdash \top} \ (\text{ax}) \qquad \text{into} \qquad \frac{}{\Gamma, \Gamma' \vdash \top} \ (\top_\text{I})$$

Alternatively, the admissibility of the rule can also be deduced from the admissibility of the cut rule (see theorem 2.2.7.5 below). □

The cut rule. A most important admissible rule is the cut rule, which states that if we can prove a formula B using a hypothesis A (thought of as a lemma used in the proof) and we can prove the hypothesis A, then we can directly prove the formula B.

Theorem 2.2.7.5 (Cut)*.* The *cut rule*

$$\frac{\Gamma \vdash A \qquad \Gamma, A, \Gamma' \vdash B}{\Gamma, \Gamma' \vdash B} \ (\text{cut})$$

is admissible.

Proof. For simplicity, we restrict ourselves to the case where the context Γ' is empty, which is not an important limitation because the exchange rule is admissible. The cut rule can be derived from the rules of implication by

$$\frac{\Gamma \vdash A \quad \dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \ (\Rightarrow_I)}{\Gamma \vdash B} \ (\Rightarrow_E) \qquad \qquad \square$$

We will see in section 2.3.2 that the above proof is not satisfactory and will provide another one, which brings much more information about the dynamics of the proofs.

Admissible rules via implication. Many rules can be proved to be admissible by eliminating provable implications:

Lemma 2.2.7.6. Suppose that the formula $A \Rightarrow B$ is provable. Then the rule

$$\frac{\Gamma \vdash A}{\Gamma \vdash B}$$

is admissible.

Proof. We have

$$\frac{\Gamma \vdash A \quad \dfrac{\dfrac{\vdots}{\vdash A \Rightarrow B}}{\Gamma \vdash A \Rightarrow B} \ (\text{wk})}{\Gamma \vdash B} \ (\Rightarrow_E) \qquad \qquad \square$$

For instance, we have seen in example 2.2.5.4 that the implication

$$(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$$

is provable. We immediately deduce:

Lemma 2.2.7.7 (Modus tollens)*.* The following two variants of the *modus tollens rule*

$$\frac{\Gamma \vdash A \Rightarrow B}{\Gamma \vdash \neg B \Rightarrow \neg A} \qquad\qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash \neg B}{\Gamma \vdash \neg A}$$

are admissible.

2.2.8 Definable connectives. A logical connective is *definable* when it can be expressed from other connectives in such a way that replacing the connective by its expression and removing the associated logical rules preserves provability.

Lemma 2.2.8.1. Negation is definable as $\neg A = A \Rightarrow \bot$.

Proof. The introduction and elimination rules of $\neg$ are derivable by

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \ (\neg_I) \quad \rightsquigarrow \quad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash A \Rightarrow \bot} \ (\Rightarrow_I)$$

$$\frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \bot} \ (\neg_E) \quad \rightsquigarrow \quad \frac{\Gamma \vdash A \Rightarrow \bot \quad \Gamma \vdash A}{\Gamma \vdash \bot} \ (\Rightarrow_E)$$

from which it follows that, given a provable formula A, the formula A' obtained from A by changing all connectives $\neg-$ into $- \Rightarrow \bot$ is provable, without using $(\neg_{\mathrm{E}})$ and $(\neg_{\mathrm{I}})$. Conversely, suppose given a formula A, such that the transformed formula A' is provable. We have to show that A is also provable, which is more subtle. In the proof of A', for each subproof of the form

$$\frac{\pi}{\Gamma \vdash B \Rightarrow \bot}$$

where the conclusion $B \Rightarrow \bot$ corresponds to the presence of $\neg B$ as a subformula of A, we can transform the proof as follows:

$$\frac{\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash B \Rightarrow \bot}}{\Gamma, B \vdash B \Rightarrow \bot}\,(\mathrm{wk}) \qquad \overline{\Gamma, B \vdash B}\,(\mathrm{ax})}{\Gamma, B \vdash \bot}\,(\Rightarrow_{\mathrm{E}})}{\Gamma \vdash \neg B}\,(\neg_{\mathrm{I}})$$

Applying this transformation enough times, we can transform the proof of A' into a proof of A. A variant of this proof is given in corollary 2.2.9.2. $\qquad \square$

Lemma 2.2.8.2. Truth is definable as $A \Rightarrow A$, for any provable formula A not involving $\top$. For instance: $\top = (\bot \Rightarrow \bot)$.

Remark 2.2.8.3. In intuitionistic logic, contrarily to what we expect from the usual de Morgan formulas, the implication is not definable as

$$A \Rightarrow B = \neg A \vee B$$

see sections 2.3.5 and 2.5.1.

2.2.9 Equivalence. We could have added to the syntax of our formulas an *equivalence* connective $\Leftrightarrow$ with associated rules

$$\frac{\Gamma \vdash A \Leftrightarrow B}{\Gamma \vdash A \Rightarrow B}\,(\Leftrightarrow_{\mathrm{E}}^{\mathrm{l}}) \qquad \frac{\Gamma \vdash A \Leftrightarrow B}{\Gamma \vdash B \Rightarrow A}\,(\Leftrightarrow_{\mathrm{E}}^{\mathrm{r}}) \qquad \frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash B \Rightarrow A}{\Gamma \vdash A \Leftrightarrow B}\,(\Leftrightarrow_{\mathrm{I}})$$

It would have been definable as

$$A \Leftrightarrow B = (A \Rightarrow B) \wedge (B \Rightarrow A)$$

Two formulas A and B are *equivalent* when $A \Leftrightarrow B$ is provable. This notion of equivalence relates in the expected way to provability:

Lemma 2.2.9.1. If A and B are equivalent then, for every context Γ, $\Gamma \vdash A$ is provable if and only if $\Gamma \vdash B$ is provable.

Proof. Immediate application of lemma 2.2.7.6. $\qquad \square$

In this way, we can give a variant of the proof of lemma 2.2.8.1:

Corollary 2.2.9.2. Negation is definable as $\neg A = (A \Rightarrow \bot)$.

Proof. We have $\neg A \Leftrightarrow (A \Rightarrow \bot)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\neg A, A \vdash \neg A}\ (\text{ax}) \quad \overline{\neg A, A \vdash A}\ (\text{ax})}{\neg A, A \vdash \bot}\ (\neg_E)}{\neg A \vdash A \Rightarrow \bot}\ (\Rightarrow_I)}{\vdash \neg A \Rightarrow A \Rightarrow \bot}\ (\Rightarrow_I) \qquad \dfrac{\dfrac{\dfrac{\overline{A \Rightarrow \bot, A \vdash A \Rightarrow \bot}\ (\text{ax}) \quad \overline{A \Rightarrow \bot, A \vdash A}\ (\text{ax})}{A \Rightarrow \bot, A \vdash \bot}\ (\Rightarrow_E)}{A \Rightarrow \bot \vdash \neg A}\ (\neg_I)}{\vdash (A \Rightarrow \bot) \Rightarrow \neg A}\ (\Rightarrow_I)}{\vdash \neg A \Leftrightarrow (A \Rightarrow \bot)}\ (\Leftrightarrow_E)$$

and we conclude using lemma 2.2.9.1. □

2.2.10 Structural rules. The rules of exchange, contraction, weakening and truth strengthening are often called *structural rules*:

$$\dfrac{\Gamma, A, B, \Gamma' \vdash C}{\Gamma, B, A, \Gamma' \vdash C}\ (\text{xch}) \qquad \dfrac{\Gamma, A, A, \Gamma' \vdash B}{\Gamma, A, \Gamma' \vdash B}\ (\text{contr})$$

$$\dfrac{\Gamma, \Gamma' \vdash B}{\Gamma, A, \Gamma' \vdash B}\ (\text{wk}) \qquad \dfrac{\Gamma, \top, \Gamma' \vdash A}{\Gamma, \Gamma' \vdash A}\ (\text{tstr})$$

We have seen in section 2.2.7 that they are admissible in our system.

Contexts as sets. The rules of exchange and contraction allow to think of contexts as sets (rather than lists) of formulas, because a set is a list "up to permutation and duplication of its elements". More precisely, given a set $\mathcal{A}$, we write $\mathcal{P}(\mathcal{A})$ for the set of subsets of $\mathcal{A}$, and $\mathcal{A}^*$ for the set of lists of elements of $\mathcal{A}$. We define an equivalence relation $\sim$ on $\mathcal{A}^*$ as the smallest equivalence relation such that

$$\Gamma, A, B, \Delta \sim \Gamma, B, A, \Delta \qquad\qquad \Gamma, A, A, \Delta \sim \Gamma, A, \Delta$$

Lemma 2.2.10.1. The function $f : \mathcal{A}^* \to \mathcal{P}(\mathcal{A})$ which to a list associates its set of elements is surjective. Moreover, given $\Gamma, \Delta \in \mathcal{A}^*$, we have $f(\Gamma) = f(\Delta)$ if and only if $\Gamma \sim \Delta$.

We could therefore have directly defined contexts to be sets of formulas, as is sometimes done, but this would be really unsatisfactory. Namely, a formula A in a context can be thought of as some kind of hypothesis which is to be proved by an auxiliary lemma and we might have twice the same formula A, but proved by different means: in this case, we would like to be able to refer to a particular instance of A (which is proved in a particular way), and we cannot do this if we have a set of hypothesis. For instance, there are intuitively two proofs of $A \Rightarrow A \Rightarrow A$: the one which uses the left A to prove A and the one which uses the right one (this will become even more striking with the Curry-Howard correspondence, see remark 4.1.7.2). However, with contexts as sets, both are the same:

$$\dfrac{\dfrac{\overline{A \vdash A}\ (\text{ax})}{A \vdash A \Rightarrow A}\ (\Rightarrow_I)}{\vdash A \Rightarrow A \Rightarrow A}\ (\Rightarrow_I)$$

A less harmful simplification which is sometimes done is to quotient by exchange only (and not contraction), in which case the contexts become multisets, see appendix A.3.5. We will refrain from doing that here as well.

Variants of the proof system. The structural rules are usually taken as "real" (as opposed to admissible) rules of the proof system. Here, we have carefully chosen the formulation of rules, so that they are admissible, but it would not hold anymore if we had used subtle variants instead. For instance, if we replace the axiom rule by

$$\frac{}{\Gamma, A \vdash A} \text{ (ax)} \qquad \text{or} \qquad \frac{}{A \vdash A} \text{ (ax)}$$

or replace the introduction rule for conjunction by

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \ (\wedge_I)$$

the structural rules are not all admissible anymore. The study of the fine structure behind this lead Girard to introduce *linear logic* [Gir87].

2.2.11 Substitution. Given formulas A and B and a variable X, we write

$$A[B/X]$$

for the *substitution* of X by B in A, i.e. the formula A where all the occurrences of X have been replaced by B. More generally, a *substitution* for A is a function which to every variable X occurring in A assigns a formula $\sigma(X)$, and we also write

$$A[\sigma]$$

for the formula A where every variable X has been replaced by $\sigma(X)$. Similarly, given a context $\Gamma = A_1, \ldots, A_n$, we define

$$\Gamma[\sigma] = A_1[\sigma], \ldots, A_n[\sigma]$$

We often write

$$[A_1/X_1, \ldots, A_n/X_n]$$

for the substitution σ such that $\sigma(X_i) = A_i$ and $\sigma(X) = X$ for X different from each X_i. It satisfies

$$A[A_1/X_1, \ldots, A_n/X_n] = A[A_1/X_1] \ldots [A_n/X_n]$$

We always suppose that, for a substitution σ, the set

$$\{X \in \mathcal{X} \mid \sigma(X) \neq X\}$$

is finite so that the substitution can be represented as the list of images of elements of this set. Provable formulas are closed under substitution:

Proposition 2.2.11.1. Given a provable sequent $\Gamma \vdash A$ and a substitution σ, the sequent $\Gamma[\sigma] \vdash A[\sigma]$ is also provable.

Proof. By induction on the proof of $\Gamma \vdash A$. □

2.3 Cut elimination

In mathematics, one often uses lemmas to show results. For instance, suppose that we want to show that 6 admits a half, i.e. there exists a number n such that $n + n = 6$. We could proceed in this way by observing that

– every even number admits a half, and

– 6 is even.

In this proof, we have used a lemma (even numbers can be halved) that we have supposed to be already proved. Of course, there was another, much shorter, proof of the fact that 6 admits a half: simply observe that $3 + 3 = 6$. We should be able to extract the second proof (giving directly 3 as a half) from the first one, by looking in details at the proof of the lemma: this process of extracting a direct proof from a proof using a lemma is called *cut elimination*. We will see that is has a number of applications and will allow us to take a "dynamic" point of view on proofs: removing cuts corresponds to "executing" proofs.

Let us illustrate how this process works in more details on the above example. We first need to make precise the notions we are using here, see section 6.6.3 for a full formalization. We say that a number m is a *half* of a number n when $m + m = n$, and the set of even numbers is defined here to be the smallest set containing 0 and such that $n + 2$ is even when n is. Moreover, our lemma is proved in this way:

Lemma 2.3.0.1. Every even number admits a half.

Proof. Suppose given an even number n. By definition of evenness, it can be of the two following forms and we can reason by induction.

– If $n = 0$ then it admits 0 as half, since $0 + 0 = 0$.

– If $n = n' + 2$ with n' even, then by induction n' admits a half m, i.e. $m + m = n'$, and therefore n admits $m + 1$ as half since

$$n = n' + 2 = (m + m) + 2 = (m + 1) + (m + 1) \qquad \square$$

In our reasoning to prove that 6 can be halved, we have used the fact that 6 is even, which we must have proved in this way:

– 6 is even because $6 = 4 + 2$ and 4 is even, where

– 4 is even because $4 = 2 + 2$ and 2 is even, where

– 2 is even because $2 = 0 + 2$ and 0 is even, where

– 0 is even by definition.

From the proof of the lemma, we know that the half of 6 is the successor of the half of 4, which is the successor of the half of 2 which is the successor of the half of 0, which is 0. Writing, as usual, $n/2$ for a half of n, we have

$$6/2 = (4/2) + 1 = (2/2) + 1 + 1 = (0/2) + 1 + 1 + 1 = 0 + 1 + 1 + 1 = 3$$

Therefore the half of 6 is 3: we have managed to extract the actual value of the half of 6 from the proofs the 6 is even and the above lemma. This example is further formalized in section 6.6.3.

2.3.1 Cuts. In logic, the use of a lemma to show a result is called a "cut". This must not be confused with the (cut) rule presented in theorem 2.2.7.5, although they are closely related. Formally, a *cut* in a proof is an elimination rule whose principal premise is proved by an introduction rule of the same connective. For instance, the following are cuts:

$$
\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A} \quad \dfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B} \; (\wedge_{\mathrm{I}})}{\Gamma \vdash A} \; (\wedge_{\mathrm{E}}^1)
\qquad\qquad
\dfrac{\dfrac{\dfrac{\pi}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B} \; (\Rightarrow_{\mathrm{I}}) \quad \dfrac{\pi'}{\Gamma \vdash A}}{\Gamma \vdash B} \; (\Rightarrow_{\mathrm{E}})
$$

The formula in the principal premise is called the *cut formula*: above, the cut formulas are respectively $A \wedge B$ and $A \Rightarrow B$. A proof containing a cut intuitively does "useless work". Namely, the one on the left starts from a proof π of A in the context Γ, which it uses to prove $A \wedge B$, from which it deduces A: in order to prove A, the proof π was already enough and the proof π' of B was entirely superfluous. Similarly, for the proof on the right, we show in π that supposing A we can prove B, and also in π' that we can prove A: we could certainly directly prove B, replacing in π all the places where the hypothesis A is used (say by an axiom) by the proof π'. For this reason, cuts are sometimes also called *detours*.

From a proof-theoretic point of view, it might seem a bit strange that someone would use such a kind of proof structure, but this is actually common in mathematics: when we want to prove a result, we often prove a lemma which is more general than the result we want to show and then deduce the result we were aiming at. One of the reasons for proceeding in this way is that we can use the same lemma to cover multiple cases, and thus have shorter proofs (not to mention that they are generally more conceptual and modular, since we can reuse the lemmas for other proofs). We will see that, however, we can always avoid using cuts in order to prove formulas. Before doing so, we first need to introduce the main technical result which allows this.

2.3.2 Proof substitution. A different kind of substitution than the one of section 2.2.11 consists in replacing some axioms in a proof by another proof. For instance, consider two proofs

$$
\pi = \dfrac{\dfrac{\dfrac{\overline{\Gamma, A \vdash A} \; (\mathrm{ax}) \quad \overline{\Gamma, A \vdash A} \; (\mathrm{ax})}{\Gamma, A, B \vdash A \wedge A} \; (\wedge_{\mathrm{I}})}{\Gamma, A \vdash B \Rightarrow A \wedge A} \; (\Rightarrow_{\mathrm{I}})
\qquad\qquad
\pi' = \dfrac{\vdots}{\Gamma \vdash A}
$$

The proof π' allows to deduce A from the hypothesis in Γ. Therefore, in the proof π, each time the hypothesis A of the context is used (by an axiom rule), we can instead use the proof π' and reprove A. Doing so, the hypothesis A in the context becomes superfluous and we can remove it. The proof resulting from this transformation is thus obtained by "re-proving" A each time we need

it instead of having it as an hypothesis:

$$\cfrac{\cfrac{\cfrac{\pi'}{\Gamma \vdash A}}{\Gamma, B \vdash A}\;(\mathrm{wk}) \qquad \cfrac{\cfrac{\pi'}{\Gamma \vdash A}}{\Gamma, B \vdash A}\;(\mathrm{wk})}{\cfrac{\Gamma, B \vdash A \wedge A}{\Gamma \vdash B \Rightarrow A \wedge A}\;(\Rightarrow_{\mathrm{I}})}\;(\wedge_{\mathrm{I}})$$

This process generalizes as follows:

Proposition 2.3.2.1. Given provable sequents

$$\cfrac{\pi}{\Gamma, A, \Gamma' \vdash B} \qquad \text{and} \qquad \cfrac{\pi'}{\Gamma \vdash A}$$

the sequent $\Gamma, \Gamma' \vdash B$ is also provable, by a proof that we write as $\pi[\pi'/A]$:

$$\cfrac{\pi[\pi'/A]}{\Gamma, \Gamma' \vdash B}$$

In other words, the (cut) rule

$$\cfrac{\Gamma \vdash A \qquad \Gamma, A, \Gamma' \vdash B}{\Gamma, \Gamma' \vdash B}(\mathrm{cut})$$

is admissible.

Proof. By induction on π. □

We will see that the admissibility of this rule is the main ingredient to prove cut elimination, thus its name.

2.3.3 Cut elimination. A logic has the *cut elimination property* when whenever a formula is provable then it is also provable with a proof which does not involve cuts: we can always avoid doing unnecessary things. This procedure was introduced by Gentzen under the name *Hauptsatz* [Gen35]. In general, we not only want to know that such a proof exists, but also to have an effective *cut elimination procedure* which transforms a proof into one without cuts. The reason for this is that we will see in section 4.1.8 that this corresponds to "executing" the proof (or the program corresponding to it): this is why Girard [Gir87] claims that

> *A logic without cut elimination is like a car without an engine.*

Although the proof obtained after eliminating cuts is "simpler" in the sense that it does not contain unnecessary steps (cuts), it cannot always be considered as "better": it is generally much bigger than the original one. The quote above explains it: think of a program computing the factorial of 1000. We see that a result can be much bigger than the program computing it [Boo84], and it can take much time to compute [Ore82].

Theorem 2.3.3.1. Intuitionistic natural deduction has the cut elimination property.

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B}\ (\Rightarrow_{\mathrm{I}}) \qquad \dfrac{\pi'}{\Gamma \vdash A}}{\Gamma \vdash B}\ (\Rightarrow_{\mathrm{E}}) \quad \rightsquigarrow \quad \dfrac{\pi[\pi'/A]}{\Gamma \vdash B}$$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A} \qquad \dfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ (\wedge_{\mathrm{I}})}{\Gamma \vdash A}\ (\wedge_{\mathrm{E}}^{\mathrm{l}}) \quad \rightsquigarrow \quad \dfrac{\pi}{\Gamma \vdash A}$$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A} \qquad \dfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ (\wedge_{\mathrm{I}})}{\Gamma \vdash B}\ (\wedge_{\mathrm{E}}^{\mathrm{r}}) \quad \rightsquigarrow \quad \dfrac{\pi'}{\Gamma \vdash B}$$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash A \vee B}\ (\vee_{\mathrm{I}}^{\mathrm{l}}) \qquad \dfrac{\pi'}{\Gamma, A \vdash C} \qquad \dfrac{\pi''}{\Gamma, B \vdash C}}{\Gamma \vdash C}\ (\vee_{\mathrm{E}}) \quad \rightsquigarrow \quad \dfrac{\pi'[\pi/A]}{\Gamma \vdash C}$$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash B}}{\Gamma \vdash A \vee B}\ (\vee_{\mathrm{I}}^{\mathrm{r}}) \qquad \dfrac{\pi'}{\Gamma, A \vdash C} \qquad \dfrac{\pi''}{\Gamma, B \vdash C}}{\Gamma \vdash C}\ (\vee_{\mathrm{E}}) \quad \rightsquigarrow \quad \dfrac{\pi''[\pi/B]}{\Gamma \vdash C}$$

Figure 2.2: Transforming proofs in NJ in order to eliminate cuts.

Proof. Suppose given a proof which contains a cut. This means that at some point in the proof we encounter one of the following situations (i.e. we have a subproof of one of the following forms), in which case we transform the proof as indicated by $\rightsquigarrow$ in figure 2.2 (we do not handle the cut on $\neg$ since $\neg A$ can be coded as $A \Rightarrow \bot$). For instance,

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma, A \vdash A}\ (\mathrm{ax}) \qquad \overline{\Gamma, A \vdash A}\ (\mathrm{ax})}{\Gamma, A \vdash A \wedge A}\ (\wedge_{\mathrm{I}})}{\Gamma \vdash A \Rightarrow A \wedge A}\ (\Rightarrow_{\mathrm{I}}) \qquad \dfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash A \wedge A}\ (\Rightarrow_{\mathrm{E}})$$

is transformed into

$$\dfrac{\dfrac{\pi}{\Gamma \vdash A} \qquad \dfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash A \wedge A}\ (\wedge_{\mathrm{I}})$$

We iterate the process on the resulting proof until all the cuts have been re-

moved.

As it can be noticed on the above example, applying the transformation $\rightsquigarrow$ might duplicate cuts: if the above proof π contained cuts, then the transformed proof contains twice the cuts of π. It is therefore not clear that the process actually terminates, whichever order we choose to eliminate cuts. We will see in section 4.2 that it indeed does, but the proof will be quite involved. It is sufficient for now to show that a particular strategy for eliminating cuts is terminating: at each step, we suppose that we eliminate a cut of highest depth, i.e. there is no cut "closer to the axioms" (for instance, we could apply the above transformation only if π has not cuts). We define the *size* $|A|$ of a formula A as its number of connectives and variables:

$$|X| = |\top| = |\bot| = 1 \qquad |A \Rightarrow B| = |A \wedge B| = |A \vee B| = 1 + |A| + |B|$$

The *degree* of a cut is the size of the cut formula (e.g. of $A \Rightarrow A \wedge A$ in the above example, whose size is $2 + 3|A|$), and the *degree* of a proof is then defined as the multiset (see appendix A.3.5) of the degrees of the cuts it contains. It can then be checked that whenever we apply $\rightsquigarrow$, the newly created cuts are of strictly lower degree than the cut we eliminated and therefore the degree of the proof decreases according to the multiset order, see appendix A.3.5. For instance, if we apply a transformation

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B} \ (\Rightarrow_{\mathrm{I}}) \qquad \cfrac{\pi'}{\Gamma \vdash A}}{\Gamma \vdash B} \ (\Rightarrow_{\mathrm{E}}) \qquad \rightsquigarrow \qquad \cfrac{\pi[\pi'/A]}{\Gamma \vdash B}$$

we suppose that π' has no cuts (otherwise the eliminated cut would not be of highest depth). The degree of the cut is $|A \Rightarrow B|$. All the cuts present in the resulting proof where already present in the original proof, except for the new cuts on A which might be created by the substitution of π' in π, which are of degree $|A| < |A \Rightarrow B|$. Since the multiset order is well-founded, see theorem A.3.5.1, the process will eventually come to an end: we cannot have an infinite sequence of $\rightsquigarrow$ transformations, chosen according to our strategy. $\quad\square$

The previous theorem states that, as long as we are interested in provability, we can restrict ourselves to cut-free proofs. This is of interest because we often have a good idea of which rules can be used in those. In particular, we have the following useful result:

Proposition 2.3.3.2. For any formula A, a cut-free proof of $\vdash A$ necessarily ends with an introduction rule.

Proof. Consider the a cut-free proof π of $\vdash A$. We reason by induction on it. This proof cannot be an axiom because the context is empty. Suppose that π ends with an elimination rule:

$$\pi = \cfrac{\vdots}{\vdash A} \ (?_{\mathrm{E}})$$

For each of the elimination rules, we observe that the principal premise is necessarily of the form $\vdash A'$, and therefore ends with an introduction rule, by

induction hypothesis. The proof is then of the form

$$
\cfrac{\cfrac{\vdots}{\vdash A'}\;(?_\mathrm{I}) \qquad \cdots}{\vdash A}\;(?_\mathrm{E})
$$

and thus contains a cut, which is impossible since we have supposed π to be cut-free. Since π cannot end with an axiom nor an elimination rule, it necessarily ends with an introduction rule. □

In the above proposition, it is crucial that we consider a formula in an empty context: a cut-free proof of $\Gamma \vdash A$ does not necessarily end with an introduction rule if Γ is arbitrary.

2.3.4 Consistency. The least one can expect from a non-trivial logical system is that not every formula is provable, otherwise the system is of no use. A logical system is *consistent* when there is at least one formula which cannot be proved in the system. Since, by $(\bot_\mathrm{E})$, one can deduce any formula from $\bot$, we have:

Lemma 2.3.4.1. The following are equivalent:

 (i) the logical system is consistent,

 (ii) the formula $\bot$ cannot be proved,

 (iii) the *principle of non-contradiction* holds: there is no formula A such that both A and $\neg A$ can be proved.

Theorem 2.3.4.2. The system NJ is consistent.

Proof. Suppose that it is inconsistent, i.e. by lemma 2.3.4.1 that it can prove $\vdash \bot$. By theorem 2.3.3.1, there is a cut-free proof of $\vdash \bot$ and, by proposition 2.3.3.2, this proof necessarily ends with an introduction rule. However, there is no introduction rule for $\bot$, contradiction. □

Remark 2.3.4.3. As a side note, we would like to point out that if we naively allowed proofs to be infinite or cyclic (i.e. contain themselves as subproofs), then the system would not be consistent anymore. For instance, we could prove $\bot$ by

$$
\pi \quad = \quad \cfrac{\cfrac{\cfrac{}{\bot \vdash \bot}\;(\mathrm{ax})}{\vdash \bot \Rightarrow \bot}\;(\Rightarrow_\mathrm{I}) \qquad \cfrac{\pi}{\vdash \bot}}{\vdash \bot}\;(\Rightarrow_\mathrm{E})
$$

(this proof is infinite in the sense that we should replace π by the proof itself above). Also, for such a proof, the cut elimination procedure would not terminate...

2.3.5 Intuitionism. We have explained in the introduction that the intuitionistic point of view on proofs is that they should be "accessible to intuition" or "constructive". This entails in particular that a proof of a disjunction $A \vee B$ should imply that one of the two formulas A or B is provable: we not only know that the disjunction is true, but we can explicitly say which one of A or B is true. This property is satisfied by the system NJ we have defined above, and this explains why we have said that it is intuitionistic:

Proposition 2.3.5.1. If a formula $A \vee B$ is provable in NJ then either A or B is provable.

Proof. Suppose that we have a proof of $A \vee B$. By theorem 2.3.3.1, we can suppose that this proof is cut-free and thus ends with an introduction rule by proposition 2.3.3.2. The proof is thus of one of the following two forms

$$\dfrac{\dfrac{\pi}{\vdash A}}{\vdash A \vee B}(\vee_{\mathrm{I}}^{\mathrm{l}}) \qquad\qquad \dfrac{\dfrac{\pi}{\vdash B}}{\vdash A \vee B}(\vee_{\mathrm{I}}^{\mathrm{r}})$$

which means that we either have a proof of A or a proof of B. $\qquad\square$

While quite satisfactory, this property means that truth in our logical systems behaves differently from the usual systems (e.g. validity in boolean models), which are called *classical* by contrast. Every formula provable in NJ is true in classical systems, but the converse is not true. One of the most striking example is the so-called *principle of excluded middle* stating that, for any formula A, the formula

$$\neg A \vee A$$

should hold. While this is certainly classically true, this cannot be proved intuitionistically for a general formula A:

Lemma 2.3.5.2. Given a propositional variable X, the formula $\neg X \vee X$ cannot be proved in NJ.

Proof. Suppose that it is provable. By proposition 2.3.5.1, either $\neg X$ or X is provable and by theorem 2.3.3.1 and proposition 2.3.3.2, we can assume that this proof is cut-free and ends with an introduction rule. Clearly, $\vdash X$ is not provable (because there is no corresponding introduction rule), so that we must have a cut-free proof of the form

$$\dfrac{\dfrac{\pi}{X \vdash \bot}}{\vdash \neg X}(\neg_{\mathrm{I}})$$

By proposition 2.2.11.1, if we had such a proof, we would in particular have one where X is replaced by $\top$:

$$\dfrac{\pi'}{\top \vdash \bot}$$

but, by proposition 2.2.7.4, we could remove $\top$ from the hypothesis and obtain a proof

$$\dfrac{\pi''}{\vdash \bot}$$

which is impossible by the consistency of NJ, see theorem 2.3.4.2. □

Of course, the above theorem does not state that, for a particular given formula A, the formula $\neg A \vee A$ is not provable. For instance, with $A = \top$, we have

$$\cfrac{\cfrac{}{\vdash \top} \ (\top_{\mathrm{I}})}{\vdash \neg\top \vee \top} \ (\vee_{\mathrm{I}}^{\mathrm{r}})$$

It however states that we cannot prove $\neg A \vee A$ without knowing the details of A. This will be studied in more detail in section 2.5, where other examples of non-provable formulas are given.

Since the excluded-middle is not provable, maybe it is false in our logic? That is not the case because we can show that the excluded-middle is not falsifiable either, since we can prove the formula $\neg\neg(\neg A \vee A)$ as follows:

$$\cfrac{\cfrac{}{\neg(\neg A \vee A) \vdash \neg(\neg A \vee A)} \ (\mathrm{ax}) \qquad \cfrac{\cfrac{\cfrac{\cfrac{}{\neg(\neg A \vee A) \vdash \neg(\neg A \vee A)} \ (\mathrm{ax}) \qquad \cfrac{\cfrac{}{\neg(\neg A \vee A), A \vdash A} \ (\mathrm{ax})}{\neg(\neg A \vee A), A \vdash \neg A \vee A} \ (\vee_{\mathrm{I}}^{\mathrm{r}})}{\neg(\neg A \vee A), A \vdash \bot} \ (\neg_{\mathrm{E}})}{\cfrac{\neg(\neg A \vee A) \vdash \neg A}{\cfrac{\neg(\neg A \vee A) \vdash \neg A \vee A}{\neg(\neg A \vee A) \vdash \bot} \ (\vee_{\mathrm{I}}^{\mathrm{l}})} \ (\neg_{\mathrm{I}})}}{\vdash \neg\neg(\neg A \vee A)} \ (\neg_{\mathrm{E}}) }{} \ (\neg_{\mathrm{I}})$$

This proof will be analyzed in more details in section 2.5.2.

A variant of the above lemma which is sometimes useful is the following one:

Lemma 2.3.5.3. Given a propositional variable X, the formula $\neg X \vee \neg\neg X$ cannot be proved in NJ.

Proof. Let us prove this in a slightly different way than in lemma 2.3.5.2. It can be proved in NJ that $\neg\top \Rightarrow \bot$:

$$\cfrac{\cfrac{\cfrac{}{\neg\top \vdash \neg\top} \ (\mathrm{ax}) \qquad \cfrac{}{\neg\top \vdash \top} \ (\top_{\mathrm{I}})}{\neg\top \vdash \bot} \ (\neg_{\mathrm{E}})}{\vdash \neg\top \Rightarrow \bot} \ (\Rightarrow_{\mathrm{I}})$$

and that $\neg\neg\bot \Rightarrow \bot$:

$$\cfrac{\cfrac{\cfrac{}{\neg\neg\bot \vdash \neg\neg\bot} \ (\mathrm{ax}) \qquad \cfrac{\cfrac{}{\neg\neg\bot, \bot \vdash \bot} \ (\mathrm{ax})}{\neg\neg\bot \vdash \neg\bot} \ (\neg_{\mathrm{I}})}{\neg\neg\bot \vdash \bot} \ (\neg_{\mathrm{E}})}{\vdash \neg\neg\bot \Rightarrow \bot} \ (\Rightarrow_{\mathrm{I}})$$

Now, suppose that we have a proof of $\neg X \vee \neg\neg X$. By proposition 2.3.5.1, either $\neg X$ or $\neg\neg X$ is provable. By proposition 2.2.11.1, either $\neg\top$ or $\neg\neg\bot$ is provable. In both cases, by $(\Rightarrow_{\mathrm{E}})$, using the above proof, $\bot$ is provable, which we know is not the case by consistency, see theorem 2.3.4.2 □

$$\dfrac{\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot_\mathrm{E}) \qquad \cdots}{\Gamma \vdash B}\ (?_\mathrm{E}) \qquad \rightsquigarrow \qquad \dfrac{\Gamma \vdash \bot}{\Gamma \vdash B}\ (\bot_\mathrm{E})$$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A \vee B} \qquad \dfrac{\pi'}{\Gamma, A \vdash C} \qquad \dfrac{\pi''}{\Gamma, B \vdash C}}{\Gamma \vdash C}\ (\vee_\mathrm{E}) \qquad \cdots}{\Gamma \vdash D}\ (?_\mathrm{E})$$

$$\updownarrow$$

$$\dfrac{\dfrac{\pi}{\Gamma \vdash A \vee B} \qquad \dfrac{\dfrac{\pi'}{\Gamma, A \vdash C} \qquad \cdots}{\Gamma, A \vdash D}\ (?_\mathrm{E}) \qquad \dfrac{\dfrac{\pi''}{\Gamma, B \vdash C} \qquad \cdots}{\Gamma, B \vdash D}\ (?_\mathrm{E})}{\Gamma \vdash D}\ (\vee_\mathrm{E})$$

Figure 2.3: Elimination of commutative cuts.

2.3.6 Commutative cuts. Are the cuts the only situations where one is doing useless work in proofs? No. It turns out that falsity and disjunction induce some more situations where we would like to eliminate "useless work". For instance, consider the following proof:

$$\dfrac{\dfrac{\dfrac{}{\bot \vdash \bot}\ (\mathrm{ax})}{\bot \vdash A \vee A}\ (\bot_\mathrm{E}) \qquad \dfrac{}{\bot, A \vdash A}\ (\mathrm{ax}) \qquad \dfrac{}{\bot, A \vdash A}\ (\mathrm{ax})}{\bot \vdash A}\ (\vee_\mathrm{E})$$

For the hypothesis $\bot$, we deduce the "general" statement that $A \vee A$ holds, from which we deduce that A holds. Clearly, we ought to be able to simplify this proof by into

$$\dfrac{\dfrac{}{\bot \vdash \bot}\ (\mathrm{ax})}{\bot \vdash A}\ (\bot_\mathrm{E})$$

where we directly prove A instead of using the "lemma" $A \vee A$ as an intermediate step. Another example of such a situation is the following one:

$$\dfrac{\dfrac{}{A, B \vee C \vdash B \vee C}\ (\mathrm{ax}) \qquad \dfrac{\dfrac{}{A, B \vee C, B \vdash A}\ (\mathrm{ax}) \quad \dfrac{}{A, B \vee C, B \vdash A}\ (\mathrm{ax})}{A, B \vee C, B \vdash A \wedge A}\ (\wedge_\mathrm{I}) \qquad \dfrac{\dfrac{}{A, B \vee C, C \vdash A}\ (\mathrm{ax}) \quad \dfrac{}{A, B \vee C, C \vdash A}\ (\mathrm{ax})}{A, B \vee C, C \vdash A \wedge A}\ (\wedge_\mathrm{I})}{\dfrac{A, B \vee C \vdash A \wedge A}{A, B \vee C \vdash A}\ (\wedge_\mathrm{E}^1)}\ (\vee_\mathrm{E})$$

Here, in a context containing A, we prove $A \wedge A$, from which we deduce A, whereas we could have directly proved A instead. This is almost a typical cut situation between the rule $(\wedge_\mathrm{I})$ and $(\wedge_\mathrm{E}^1)$, except that we cannot eliminate the cut because the two rules are separated by the intermediate rule $(\vee_\mathrm{E})$. In order for the system to have nice properties, we should thus add to the usual cut-elimination rules the rules of figure 2.3, where $(?_\mathrm{E})$ stands for an arbitrary elimination rule. Those rules eliminate what we call *commutative cuts*, see [Gir89, section 10.3].

2.4 Proof search

An important question is whether there is an automated procedure in order to perform proof-search in NJ, i.e. answer the question:

Is a given sequent $\Gamma \vdash A$ provable?

In general, the answer is yes, but the complexity is hard. In order to do so, the basic idea of course consists in trying to construct a proof derivation whose conclusion is our sequent, from bottom up.

2.4.1 Reversible rules. A rule is *reversible* when, if its conclusion is provable, then its hypothesis are provable. Such rules are particularly convenient in order to search for proofs since we know that we can always apply them: if the conclusion sequent was provable then the hypothesis still are. For instance, the rule

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \; (\vee_{\mathrm{I}}^{\mathrm{l}})$$

is not reversible: if, while searching for a proof of $\Gamma \vdash A \vee B$, we apply it, we might have to backtrack in the case where $\Gamma \vdash A$ is not provable, since maybe $\Gamma \vdash B$ was provable instead, the most extreme example being

$$\frac{\begin{array}{c} \vdots \\ \hline \vdash \bot \end{array}}{\vdash \bot \vee \top} \; (\vee_{\mathrm{I}}^{\mathrm{l}})$$

where we have picked the wrong branch of the disjunction and try to prove $\bot$, whereas $\top$ was directly provable. On the contrary, the rule

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \; (\wedge_{\mathrm{I}})$$

is reversible: during proof search, we can apply it without regretting our choice.

Proposition 2.4.1.1. In NJ, the reversible rules are (ax), ($\Rightarrow_{\mathrm{I}}$), ($\wedge_{\mathrm{I}}$), ($\top_{\mathrm{I}}$) and ($\neg_{\mathrm{I}}$).

Proof. Consider the case of ($\Rightarrow_{\mathrm{I}}$), the other cases being similar. In order to show that this rule (recalled on the left) is reversible, we have to show that if the conclusion is provable then the premise also is, i.e. that the rule on the right is admissible:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \; (\Rightarrow_{\mathrm{I}}) \qquad\qquad \frac{\Gamma \vdash A \Rightarrow B}{\Gamma, A \vdash B}$$

Suppose that we have a proof π of the conclusion $\Gamma \vdash A \Rightarrow B$. We can construct a proof of $\Gamma, A \vdash B$ by

$$\frac{\dfrac{\dfrac{\pi}{\Gamma \vdash A \Rightarrow B}}{\Gamma, A \vdash A \Rightarrow B} \; (\mathrm{wk}) \qquad \dfrac{}{\Gamma, A \vdash A} \; (\mathrm{ax})}{\Gamma, A \vdash B} \; (\Rightarrow_{\mathrm{E}}) \qquad\qquad \square$$

For instance, we want to prove the formula $X \Rightarrow Y \Rightarrow X \wedge Y$. We can try to apply the reversible rules as long as we can, and indeed, we end up with a proof:

$$\cfrac{\cfrac{\cfrac{\overline{X, Y \vdash X} \; (\mathrm{ax}) \qquad \overline{X, Y \vdash Y} \; (\mathrm{ax})}{X, Y \vdash X \wedge Y} \; (\wedge_{\mathrm{I}})}{X \vdash Y \Rightarrow X \wedge Y} \; (\Rightarrow_{\mathrm{I}})}{\vdash X \Rightarrow Y \Rightarrow X \wedge Y} \; (\Rightarrow_{\mathrm{I}})$$

2.4.2 Proof search. Proof search can be automated in NJ: there is an algorithm which, given a sequent, determines whether it is provable or not. We describe here such an algorithm where, for simplicity, we restrict ourselves here to the implicational fragment (formulas are built out of variables and implication, and the rules are (ax), ($\Rightarrow_{\mathrm{E}}$) and ($\Rightarrow_{\mathrm{I}}$)).

Suppose that we are trying to determine whether a given sequent $\Gamma \vdash A$ is provable. It can be observed that, depending on the formula A (which is either of the form $B \Rightarrow C$ or a variable X), we can always look for proofs of the following form:

(a) $\Gamma \vdash B \Rightarrow C$: the last rule is

$$\frac{\Gamma, B \vdash C}{\Gamma \vdash B \Rightarrow C} \; (\Rightarrow_{\mathrm{I}})$$

and we look for a proof of $\Gamma, B \vdash C$,

(b) $\Gamma \vdash X$: the proof ends with

$$\cfrac{\cfrac{\cfrac{\overline{\Gamma \vdash A_1 \Rightarrow A_2 \Rightarrow \ldots \Rightarrow A_n \Rightarrow X} \; (\mathrm{ax}) \qquad \cfrac{\vdots}{\Gamma \vdash A_1}}{\Gamma \vdash A_2 \Rightarrow \ldots \Rightarrow A_n \Rightarrow X} \; (\Rightarrow_{\mathrm{E}}) \qquad \cfrac{\vdots}{\Gamma \vdash A_2}}{\vdots} \; (\Rightarrow_{\mathrm{E}})}{\cfrac{\Gamma \vdash A_n \Rightarrow X \qquad\qquad\qquad (\Rightarrow_{\mathrm{E}}) \qquad \cfrac{\vdots}{\Gamma \vdash A_n}}{\Gamma \vdash X}} \; (\Rightarrow_{\mathrm{E}})$$

where the particular case $n = 0$ is

$$\frac{}{\Gamma \vdash X} \; (\mathrm{ax})$$

and we thus try to find in the context a formula of the form

$$A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow X$$

such that all the $\Gamma \vdash A_i$ are provable.

Namely, the first case is justified by the fact that ($\Rightarrow_{\mathrm{I}}$) is reversible so that it can always be applied first, and the second one by the fact that we can look for cut-free proofs (theorem 2.3.3.1) so that we can restrict to the cases where the rules ($\Rightarrow_{\mathrm{E}}$) have a principal premise which is a rule (ax) or ($\Rightarrow_{\mathrm{E}}$), but not ($\Rightarrow_{\mathrm{I}}$).

This suggests the following procedure to determine the provability of a given sequent $\Gamma \vdash A$:

– if A is of the form $B \Rightarrow C$, we recursively try to prove the sequent $\Gamma, B \vdash C$,

– if A is a variable X, we try to find in the context Γ a formula of the form $A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow X$ such that all the sequents $\Gamma \vdash A_i$ are provable, which can be tested recursively.

The problem with this procedure is that it might not terminate. For instance, given the sequent $X \Rightarrow X \vdash X$, the procedure will loop, trying to construct an infinite proof tree of the form

$$\cfrac{\cfrac{}{\Gamma \vdash X \Rightarrow X}\ (\text{ax}) \quad \cfrac{\cfrac{}{\Gamma \vdash X \Rightarrow X}\ (\text{ax}) \quad \cfrac{\cfrac{}{\Gamma \vdash X \Rightarrow X}\ (\text{ax}) \quad \cfrac{\vdots}{\Gamma \vdash X}\ (\Rightarrow_{\text{E}})}{\Gamma \vdash X}\ (\Rightarrow_{\text{E}})}{\Gamma \vdash X}\ (\Rightarrow_{\text{E}})}{\Gamma \vdash X}\ (\Rightarrow_{\text{E}})$$

In order to prevent this kind of loops, we should ensure that, whenever we are trying to construct a proof of a sequent $\Gamma \vdash A$, we never try to construct again a proof of $\Gamma \vdash A$ at a later stage, and this is easily done by remembering all the sequents encountered during proof search. An actual implementation is provided in figure 2.4, where we use a list `seen` to remember the sequents that we have already seen, a sequent being encoded as a pair consisting of a context (a list of formulas) and a formula.

Writing Γ for the context $X \Rightarrow Y, (X \Rightarrow Y) \Rightarrow X$, our algorithm will find that $\Gamma \vdash Y$ is provable because there is the proof

$$\cfrac{\cfrac{}{\Gamma \vdash X \Rightarrow Y}\ (\text{ax}) \quad \cfrac{\cfrac{}{\Gamma \vdash (X \Rightarrow Y) \Rightarrow X}\ (\text{ax}) \quad \cfrac{\cfrac{\cfrac{}{\Gamma, X \vdash X \Rightarrow Y}\ (\text{ax}) \quad \cfrac{}{\Gamma, X \vdash X}\ (\text{ax})}{\Gamma, X \vdash Y}\ (\Rightarrow_{\text{E}})}{\Gamma \vdash X \Rightarrow Y}\ (\Rightarrow_{\text{I}})}{\Gamma \vdash X}\ (\Rightarrow_{\text{E}})}{\Gamma \vdash Y}\ (\Rightarrow_{\text{E}})$$

Note that we are trying to prove Y twice during the proof search, but this is authorized because this is done in different contexts (respectively in the contexts Γ and Γ, X). As it can be observed in the above example, when looking for a proof a sequent $\Gamma \vdash A$, the contexts can grow during proof search. Termination is however still guaranteed because it can be shown that all the formulas that we add to the context are strict subformulas of the original formula A, and there are only a finite number of those. The algorithm can be shown to be in PSPACE (i.e. it requires an amount of memory which is polynomial in the size of its input) and the problem is actually PSPACE-complete (it is harder than any other problem in PSPACE, which in particular implies that it is harder than any problem in NP), see [Sta79] and [SU06, section 6.6]. Other methods for performing proof search in intuitionistic logic are presented in section 2.6.5.

2.5 Classical logic

As we have seen in section 2.3.5, not all the formulas that we expect to hold in logic are provable in intuitionistic logic, such as the excluded middle (lemma 2.3.5.2).

```
(** Formulas. *)
type t =
  | Var of string
  | Imp of t * t

(** Split arguments and target of implications. *)
let rec split_imp = function
  | Var x -> [], Var x
  | Imp (a, b) ->
    let args, tgt = split_imp b in
    a::args, tgt

(** Determine whether a sequent is provable in a given context. *)
let rec provable seen env a =
  not (List.mem (env,a) seen) &&
  let seen = (env,a)::seen in
  match a with
  | Var x ->
    List.exists (fun a ->
        let args, b = split_imp a in
        b = Var x && List.for_all (provable seen env) args
      ) env
  | Imp (a, b) -> provable seen (a::env) b

let provable = provable []
```

Figure 2.4: Deciding provability in intuitionistic logic.

In contrast, the usual notion of validity (e.g. coming from boolean models) is called *classical logic*. If classical logic is closer to the usual intuition of validity, the main drawback for us is that this logic is not *constructive*, in the sense that we cannot necessarily extract witnesses from proofs: if we have proved $\neg A \lor A$, we do not necessarily know which one of $\neg A$ or A actually holds.

A well-known typical classical reasoning is the following. We want to prove that there exist two irrational numbers a and b such that a^b is rational. We know that $\sqrt{2}$ is irrational: if $\sqrt{2} = p/q$ then $p^2 = 2q^2$, but the number of prime factors is even on the left and odd on the right. Reasoning using the excluded middle, we know that the number $\sqrt{2}^{\sqrt{2}}$ is either rational or irrational:

– if it is rational, we conclude with $a = b = \sqrt{2}$,

– otherwise, we take $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$, and we have $a^b = 2$ which concludes the proof.

We have been able to prove the property, but we are not able to exhibit a concrete value for a and b.

From the proof-as-program correspondence, the excluded middle is also quite puzzling. Suppose that we are in a logic rich enough to encode Turing machines (or, equivalently, execute a program in a usual programming language) and that we have a predicate $\mathrm{Halts}(M)$ which holds when M is halting (you should find this quite plausible after having read chapter 6). In classical logic, the formula

$$\neg\,\mathrm{Halts}(M) \lor \mathrm{Halts}(M)$$

holds for every Turing machine M, which seems to mean that we should be able to decide whether a Turing machine is halting or not, but there is no hope of finding such an algorithm since Turing has shown that the halting problem is undecidable [Tur37b].

2.5.1 Axioms for classical logic. A logical system for classical logic, called NK (for *K*lassical *N*atural deduction), can be obtained from NJ (figure 2.1) by adding a new rule corresponding to the excluded middle

$$\frac{}{\Gamma \vdash \neg A \lor A}\ (\mathrm{lem})$$

In this sense, the excluded middle is the only thing which is missing in intuitionistic logic to be classical. This is shown in theorems 2.5.6.1 and 2.5.6.5.

In fact, excluded middle is not the only possible choice, and other equivalent axioms can be added instead. Most of those axioms correspond to usual reasoning patterns, which have been known for a long time, and thus bear latin names.

Theorem 2.5.1.1. The following principles are equivalent in NJ:

  (i) *excluded middle*, also called *tertium non datur*:

$$\neg A \lor A$$

  (ii) *double-negation elimination* or *reductio ad absurdum*:

$$\neg\neg A \Rightarrow A$$

(iii) *contraposition*:
$$(\neg B \Rightarrow \neg A) \Rightarrow (A \Rightarrow B)$$

(iv) *counter-example principle*:
$$\neg(A \Rightarrow B) \Rightarrow A \wedge \neg B$$

(v) *Peirce's law*:
$$((A \Rightarrow B) \Rightarrow A) \Rightarrow A$$

(vi) *Clavius' law* or *consequentia mirabilis*:
$$(\neg A \Rightarrow A) \Rightarrow A$$

(vii) *Tarski's formula*:
$$A \vee (A \Rightarrow B)$$

(viii) one of the following *de Morgan laws*:
$$\neg(\neg A \wedge \neg B) \Rightarrow A \vee B$$
$$\neg(\neg A \vee \neg B) \Rightarrow A \wedge B$$

(ix) *material implication*:
$$(A \Rightarrow B) \Rightarrow (\neg A \vee B)$$

(x) $\Rightarrow/\vee$ *distributivity*:
$$(A \Rightarrow (B \vee C)) \Rightarrow ((A \Rightarrow B) \vee C)$$

By "equivalent" we mean here that if we suppose that one holds for every formulas A, B and C then the other one also holds for every formulas A, B and C, and conversely.

Proof. We only show here the equivalence between the first two, the other ones being left as an exercise. Supposing that the excluded middle holds, we can show reductio ad absurdum by

$$
\cfrac{
\cfrac{}{\neg\neg A \vdash \neg A \vee A}
\quad
\cfrac{
\cfrac{\overline{\neg\neg A, \neg A \vdash \neg\neg A}\ (\text{ax}) \quad \overline{\neg\neg A, \neg A \vdash \neg A}\ (\text{ax})}{\neg\neg A, \neg A \vdash A}\ (\neg_{\text{E}})
\quad
\cfrac{}{\neg\neg A, A \vdash A}\ (\text{ax})
}{\neg\neg A \vdash A}\ (\vee_{\text{E}})
}{
\cfrac{\vdash \neg\neg A \Rightarrow A}{}
}\ (\Rightarrow_{\text{I}})
$$

$$(2.2)$$

Supposing that reductio ad absurdum holds, we can show the excluded middle by

$$
\cfrac{
\cfrac{}{\vdash \neg\neg(\neg A \vee A) \Rightarrow (\neg A \vee A)}
\quad
\cfrac{\pi}{\vdash \neg\neg(\neg A \vee A)}
}{\vdash \neg A \vee A}\ (\vee_{\text{E}})
$$

$$(2.3)$$

where π is the proof of $\neg\neg(\neg A \vee A)$ given on page 63. $\square$

Remark 2.5.1.2. One should be careful about the quantifications over formulas involved in theorem 2.5.1.1. In order to illustrate this, let us detail the equivalence between excluded middle and reductio ad absurdum. We say that a formula A is *decidable* when $\neg A \vee A$ holds and *stable* when $\neg\neg A \Rightarrow A$ holds. The derivation (2.2) shows that every decidable formula is stable, but the converse does not hold: the derivation (2.3) only shows that A is decidable when $\neg A \vee A$ (as opposed to A) is stable. In fact a concrete example of a formula which is stable but not decidable can be given by taking $A = \neg X$: the formula $\neg\neg\neg X \Rightarrow \neg X$ holds (lemma 2.5.9.4), but $\neg\neg X \vee \neg X$ cannot be proved (lemma 2.3.5.3). Thus, it is important to note that theorem 2.5.1.1 does not say that a formula is stable if and only if it is decidable, but rather that every formula is stable if and only if every formula is decidable.

Among those axioms, Pierce's law is less natural than others but has the advantage of requiring only implication, so that it still makes sense in some small fragments of logic such as implicational logic. Also note that the fact that material implication occurs in this list means that $A \Rightarrow B$ is not equivalent to $\neg A \vee B$ in NJ, in contrast to NK. For each of these axioms, we could add more or less natural forms of rules. For instance, the law of the excluded middle can also be implemented by the nicer looking rule

$$\frac{\Gamma, \neg A \vdash B \qquad \Gamma, A \vdash B}{\Gamma \vdash B} \; (\text{lem})$$

similarly, reductio ad absurdum can be implemented by one of the following rules

$$\frac{}{\Gamma \vdash \neg\neg A \Rightarrow A} \qquad \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A} \; (\neg\neg_{\text{E}}) \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A} \qquad \frac{\Gamma, \neg A \vdash A}{\Gamma \vdash A}$$

Since classical logic is obtained from intuitionistic by adding axioms, it is obvious that

Lemma 2.5.1.3. An intuitionistic proof is a valid classical proof.

We have seen that the converse does not hold (lemma 2.3.5.2), but we will see in section 2.5.9 that we can still embed classical proofs in intuitionistic logic.

2.5.2 The intuition behind classical logic. Let us try to give some proof theoretic intuition about how classical logic works.

Proof irrelevance. We have already mentioned that we can interpret a formula as a set $[\![A]\!]$, intuitively corresponding to all the proofs of A, and implications as function spaces: $[\![A \Rightarrow B]\!] = [\![A]\!] \to [\![B]\!]$. In this interpretation, $\bot$ of course corresponds to the empty set since we do not expect to have a proof of it: $[\![\bot]\!] = \emptyset$. Now, given a formula A, its negation $\neg A = (A \Rightarrow \bot)$ is interpreted as the set of functions from $[\![A]\!]$ to $\emptyset$:

   – if $[\![A]\!]$ is non-empty, the set $[\![\neg A]\!] = [\![A]\!] \to \emptyset$ is empty,

   – if $[\![A]\!]$ is empty, the set $[\![\neg A]\!] = \emptyset \to \emptyset$ contains exactly one element.

The last point might seem surprising, but if we think hard about it it makes sense. For instance, in set theory, a function $f : [\![A]\!] \to [\![B]\!]$ is usually defined as a relation $f \subseteq [\![A]\!] \times [\![B]\!]$ which satisfies some properties, expressing that each element of $[\![A]\!]$ should have exactly one image in $[\![B]\!]$. Now, when both the sets are empty, we are looking for a relation $f \subseteq \emptyset \times \emptyset$ and there is exactly one such relation: the empty set (which trivially satisfies the axioms for functions).

Applying twice the reasoning above, we get that

– if $[\![A]\!]$ is non-empty, $[\![\neg\neg A]\!]$ contains exactly one element,

– if $[\![A]\!]$ is empty, $[\![\neg\neg A]\!]$ is empty.

In other words, $\neg\neg A$ can be seen as the formula A where the only thing which matters is not all the proofs of A (i.e. the elements of $[\![A]\!]$), but only whether there exists a proof of A or not, since we have reduced its contents to at most one point. For this reason, doubly negated formulas are sometimes said to be *proof irrelevant*: again, the actual proof does not matter, only its existence. For instance, we now understand why

$$\neg\neg(\neg A \vee A)$$

is provable intuitionistically (see page 63): it states that it is true that there exists a proof of $\neg A$ or a proof of A, as opposed to $\neg A \vee A$ which states that we have a proof of $\neg A$ or a proof of A. From this point of view, the classical axiom

$$\neg\neg A \Rightarrow A$$

now seems like deep magic: it means that if we know that there exists a proof of A, we can actually extract a proof of A. This can only be true if we assume that there can be at most one proof for a formula, i.e. formulas are interpreted as booleans and not sets (see section 2.5.4 for a logical point of view on this). This also explains why we can actually embed classical logic into intuitionistic logic by double-negating formulas, see section 2.5.9: if we are only interested in their existence, intuitionistic proofs behave classically.

Resetting proofs. Let us give another, more operational, point of view on the axiom $\neg\neg A \Rightarrow A$. We have mentioned that it is equivalent to having the rule

$$\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A} \ (\neg\neg\text{E})$$

so that when searching for a proof of A, we can instead prove $\neg\neg A$. What do we gain in doing so? At first it does not seem much, since we can go back to proving A:

$$\frac{\dfrac{}{\Gamma, \neg A \vdash \neg A} \ (\text{ax}) \qquad \dfrac{\vdots}{\Gamma, \neg A \vdash A}}{\dfrac{\dfrac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash \neg\neg A} \ (\neg\text{I})}{}} \ (\neg\text{E})$$

But there is one difference: we now have the additional hypothesis $\neg A$ in our context, and we can use it at any point in the proof to go back to proving A

instead of the current goal B, while keeping the current context:

$$
\cfrac{
\cfrac{
\dfrac{}{\Gamma', \neg A \vdash \neg A} \text{ (ax)} \qquad
\dfrac{\vdots}{\Gamma', \neg A \vdash A}
}{\Gamma', \neg A \vdash \bot} \; (\neg_{\mathrm{E}})
}{\Gamma', \neg A \vdash B} \; (\bot_{\mathrm{E}})
$$

In other words, we can "reset proofs" during proof search, i.e. we can implement the following behavior (up to minor details such as weakening):

$$
\cfrac{
\cfrac{
\dfrac{\vdots}{\Gamma' \vdash A}
}{\Gamma' \vdash B} \; \text{(reset)}
}{
\cfrac{\vdots}{\Gamma \vdash A}
}{}
$$

Note that we keep the context Γ' after the reset.

Now, let us show how we can use this to prove $\neg A \vee A$. When faced with the disjunction, we choose the left branch, i.e. prove $\neg A$, which by $(\neg_{\mathrm{I}})$ amounts to proving $\bot$, supposing A as hypothesis. Instead of going on and proving $\bot$, which is quite hopeless, we use our reset mechanism and go back to proving $\neg A \vee A$: while doing so we have kept A as hypothesis! So, this time we chose to prove A, which can be done by an axiom. If we think of reset as the possibility of "going back in time" and changing one's mind, this proof implements the following conversation between us, trying to build the proof, and an opponent trying to prove us wrong:

— Show me the formula $\neg A \vee A$.

— Ok, I will show that $\neg A$ holds.

— Here is a proof π of A, show me how to deduce $\bot$.

— Actually, I changed my mind, I will prove A, here is the proof: π.

The formal proof goes like this

$$
\cfrac{
\cfrac{
\cfrac{
\dfrac{}{A \vdash A} \text{ (ax)}
}{A \vdash \neg A \vee A} \; (\vee_{\mathrm{I}}^{\mathrm{r}})
}{A \vdash \bot} \; \text{(reset)}
}{
\cfrac{}{\vdash \neg A} \; (\neg_{\mathrm{I}})
}{\vdash \neg A \vee A} \; (\vee_{\mathrm{I}}^{\mathrm{l}})
$$

In more details, the proof begins by proving $\neg\neg(\neg A \vee A)$ instead of $\neg A \vee A$ and then proceeds as in the proof given on page 63. This idea of resetting will be explored again, in a different form, in section 4.6.

$$\frac{}{\Gamma, A, \Gamma' \vdash A, \Delta} \ (\text{ax})$$

$$\frac{\Gamma \vdash A \Rightarrow B, \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash B, \Delta} \ (\Rightarrow_{\text{E}}) \qquad\qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \ (\Rightarrow_{\text{I}})$$

$$\frac{\Gamma \vdash A \wedge B, \Delta}{\Gamma \vdash A, \Delta} \ (\wedge_{\text{E}}^{\text{l}}) \qquad \frac{\Gamma \vdash A \wedge B, \Delta}{\Gamma \vdash B, \Delta} \ (\wedge_{\text{E}}^{\text{r}}) \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \ (\wedge_{\text{I}})$$

$$\frac{}{\Gamma \vdash \top, \Delta} \ (\top_{\text{I}})$$

$$\frac{\Gamma \vdash A \vee B, \Delta \qquad \Gamma, A \vdash C, \Delta \qquad \Gamma, B \vdash C, \Delta}{\Gamma \vdash C, \Delta} \ (\vee_{\text{E}}) \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \ (\vee_{\text{I}})$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \bot, \Delta} \ (\bot_{\text{I}})$$

$$\frac{\Gamma \vdash \neg A, \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash \bot, \Delta} \ (\neg_{\text{E}}) \qquad\qquad \frac{\Gamma, A \vdash \bot, \Delta}{\Gamma \vdash \neg A, \Delta} \ (\neg_{\text{I}})$$

<div align="center">structural rules:</div>

$$\frac{\Gamma \vdash \Delta, A, B, \Delta'}{\Gamma \vdash \Delta, B, A, \Delta'} \ (\text{xch}_{\text{R}}) \qquad\qquad \frac{\Gamma \vdash \Delta, \Delta'}{\Gamma \vdash \Delta, A, \Delta'} \ (\text{wk}_{\text{R}})$$

$$\frac{\Gamma \vdash \Delta, A, A, \Delta'}{\Gamma \vdash \Delta, A, \Delta'} \ (\text{contr}_{\text{R}}) \qquad\qquad \frac{\Gamma \vdash \Delta, \bot, \Delta'}{\Gamma \vdash \Delta, \Delta'} \ (\bot_{\text{R}})$$

<div align="center">Figure 2.5: NK: rules of classical natural deduction.</div>

2.5.3 A variant of natural deduction. The presentation given in section 2.5.1 is not very "canonical" in the sense that it amounts to randomly add an axiom from the list given in theorem 2.5.1.1. We would like to present another approach which consists in slightly changing the calculus, and allows for a much more pleasant proof system. We now consider sequents of the form

$$\Gamma \vdash \Delta$$

where both Γ and Δ are contexts. Such a sequent should be read as "supposing all the formula in Γ, I can prove some formula in Δ". This is a generalization of previous sequents, where Δ was restricted to exactly one formula. The rules for this sequent calculus are given in figure 2.5. In order to simplify the presentation, we consider here that the formulas of Δ can be explicitly permuted, duplicated, and so on, using the structural rules (xch$_{\text{R}}$), (wk$_{\text{R}}$), (contr$_{\text{R}}$) and ($\bot_{\text{R}}$), which we generally leave implicit in examples. Those rules are essentially the same as those for NJ, with contexts added on the right, except for the rules ($\vee_{\text{I}}^{\text{l}}$) and

($\vee_\text{I}^\text{r}$), which are now combined into the rule

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \; (\vee_\text{I})$$

In order to prove a disjunction $A \vee B$, we do not have to choose anymore if we want to prove A or B: we can try to prove both at the same time. This means that there can be some "exchange of information" between the proofs of A and of B (via the context Γ). For instance, we have that the excluded middle can be proved by

$$\frac{\dfrac{\overline{A \vdash A, \bot} \; (\text{ax})}{A \vdash \bot, A} \; (\text{xch}_\text{R})}{\dfrac{\vdash \neg A, A}{\vdash \neg A \vee A} \; (\vee_\text{I})} \; (\neg_\text{I})$$

Note that the formula A in the context, obtained from the $\neg A$, is used in the axiom in order to prove the other A. Similarly, double negation elimination is proved by

$$\cfrac{\cfrac{\cfrac{\overline{\neg\neg A \vdash \neg\neg A, A} \; (\text{ax}) \qquad \cfrac{\cfrac{\overline{\neg\neg A, A \vdash A} \; (\text{ax})}{\neg\neg A, A \vdash \bot, A} \; (\text{wk}_\text{R})}{\neg\neg A \vdash \neg A, A} \; (\neg_\text{I})}{\neg\neg A \vdash \bot, A}}{\neg\neg A \vdash A, A} \; (\bot_\text{E})}{\cfrac{\neg\neg A \vdash A}{\vdash \neg\neg A \Rightarrow A} \; (\Rightarrow_\text{I})} \; (\text{contr}_\text{R})}{} \; (\neg_\text{E})$$

Again, instead of proving A, we decide to either prove $\bot$ or A.

The expected elimination rule for the constant $\bot$ (shown on the left) is not present, but it can be derived (as shown on the right):

$$\frac{\Gamma \vdash \bot, \Delta}{\Gamma \vdash A, \Delta} \; (\bot_\text{E}) \qquad\qquad \frac{\dfrac{\Gamma \vdash \bot, \Delta}{\Gamma \vdash \Delta} \; (\bot_\text{R})}{\Gamma \vdash A, \Delta} \; (\text{wk}_\text{R})$$

In fact the constant $\bot$ is now superfluous, since one can convince himself that proving $\bot$ amounts to proving the empty sequent Δ.

2.5.4 Cut-elimination in classical logic. Classical logic also does have the cut-elimination property, see section 2.3.3, although this is more subtle to show than in the case of intuitionistic logic due to the presence of structural rules. In particular, in addition to the usual cut elimination steps, we need to add rules making elimination rules "commute" with structural rules: namely, an introduction and the corresponding elimination rules can be separated by structural

rules. For instance, suppose that we want to eliminate the following "cut":

$$\cfrac{\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash A} \qquad \cfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ (\wedge_{\mathrm{I}})}{\Gamma \vdash A \wedge B, C}\ (\mathrm{wk_R})}{\Gamma \vdash A, C}\ (\wedge_{\mathrm{E}}^1)$$

We first need to make the elimination rule for conjunction commute with the weakening:

$$\cfrac{\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash A} \qquad \cfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ (\wedge_{\mathrm{I}})}{\Gamma \vdash A}\ (\wedge_{\mathrm{E}}^1)}{\Gamma \vdash A, C}\ (\mathrm{wk_R})$$

and then we can finally properly eliminate the cut:

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash A}\ (\wedge_{\mathrm{E}}^1)}{\Gamma \vdash A}}{\Gamma \vdash A, C}\ (\mathrm{wk_R})$$

Another surprising phenomenon was observed by Lafont [Gir89, section B.1]. Depending on the order in which we eliminate cuts, the following proof

$$\cfrac{\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash \neg C, A}\ (\mathrm{wk_R}) \qquad \cfrac{\cfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash C, B}\ (\mathrm{wk_R})}{\Gamma \vdash \bot, A, B}\ (\neg_{\mathrm{E}})}{\Gamma \vdash A, B}\ (\bot_{\mathrm{R}})$$

both cut-eliminates to

$$\cfrac{\cfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash A, B}\ (\mathrm{wk_R}) \qquad \text{and} \qquad \cfrac{\cfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A, B}\ (\mathrm{wk_R})$$

This is sometimes called *Lafont's critical pair*. We like to identify proofs up to cut elimination (much more on this in chapter 4) and therefore those two proofs should be considered as being "the same". In particular, when both π and π' are proofs of $\Gamma \vdash A$, i.e. $A = B$, this forces us to identify the two proofs

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash A, A}\ (\mathrm{wk_R})}{\Gamma \vdash A}\ (\mathrm{contr_R}) \qquad \text{and} \qquad \cfrac{\cfrac{\cfrac{\pi'}{\Gamma \vdash A}}{\Gamma \vdash A, A}\ (\mathrm{wk_R})}{\Gamma \vdash A}\ (\mathrm{contr_R})$$

and thus to identify the two proofs π and π'. More generally, by similar reasoning, any two proofs of a same sequent $\Gamma \vdash \Delta$ should be identified. Cuts can hurt! This gives another, purely logical, explanation of why classical logic is "proof irrelevant", as already mentioned in section 2.5.2: up to cut-elimination, there is at most one proof of a given sequent.

2.5.5 De Morgan laws. In classical logic, the well-known *de Morgan laws* hold:

$$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B \qquad \neg\top \Leftrightarrow \bot \qquad A \Rightarrow B \Leftrightarrow \neg A \vee B$$
$$\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B \qquad \neg\bot \Leftrightarrow \top \qquad \neg\neg A \Leftrightarrow A$$

Definable connectives. Because of these laws, many connectives are superfluous. For instance, classical logic can be axiomatized with $\Rightarrow$ and $\bot$ as the only connectives, since we can define

$$A \vee B = \neg A \Rightarrow B \quad A \wedge B = \neg(A \Rightarrow \neg B) \quad \neg A = A \Rightarrow \bot \quad \top = \bot \Rightarrow \bot$$

and the logical system can be reduced to the following four rules:

$$\frac{}{\Gamma, A, \Gamma' \vdash A, \Delta} \text{ (ax)} \qquad\qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \bot, \Delta} \ (\bot_{\text{I}})$$

$$\frac{\Gamma \vdash A \Rightarrow B, \Delta \quad \Gamma \vdash A, \Delta}{\Gamma \vdash B, \Delta} \ (\Rightarrow_{\text{E}}) \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \ (\Rightarrow_{\text{I}})$$

together with the four structural rules. Several other choices of connectives are possible.

Clausal form. It is natural to consider the equivalence relation on formulas which identifies any two formulas A and B such that $A \Leftrightarrow B$. The de Morgan laws can be used to rewrite every formula into a canonical representative of its equivalence class induced by this equivalence relation. We first need to introduce some classes of formulas.

A *literal L* is either a variable or a negated variable:

$$L ::= X \mid \neg X$$

A *clause C* is a disjunction of literals:

$$C ::= L \mid C \vee C \mid \bot$$

A formula A is in *clausal form* or in *conjunctive normal form* when it is a conjunction of clauses:

$$A ::= C \mid A \wedge A \mid \top$$

Proposition 2.5.5.1. Every formula is equivalent to one in clausal form.

One way to show this result is to use the de Morgan laws, as well as usual intuitionistic laws (section 2.2.5), in order to push negations toward variables and disjunctions below conjunctions, i.e. we replace subformulas according to the following rules, until no rule applies:

$$\neg(A \wedge B) \rightsquigarrow \neg A \vee \neg B \qquad\qquad \neg\top \rightsquigarrow \bot$$
$$\neg(A \vee B) \rightsquigarrow \neg A \wedge \neg B \qquad\qquad \neg\bot \rightsquigarrow \top$$
$$(A \wedge B) \vee C \rightsquigarrow (A \vee C) \wedge (B \vee C) \qquad \top \vee C \rightsquigarrow \top$$
$$A \vee (B \wedge C) \rightsquigarrow (A \vee B) \wedge (A \vee C) \qquad A \vee \top \rightsquigarrow \top$$
$$A \Rightarrow B \rightsquigarrow \neg A \vee B \qquad\qquad \neg\neg A \rightsquigarrow A$$

Those rules rewrite formulas into classically equivalent ones, since those are instances of de Morgan laws. However, it is not clear that the process terminates. It does, but it is not efficient, and we will see below a better way to rewrite a formula in clausal form.

Example 2.5.5.2. A clausal form of $(X \Rightarrow Y) \Rightarrow (Y \Rightarrow Z)$ can be computed by

$$
\begin{aligned}
\neg(\neg X \vee Y) \vee (\neg Y \vee Z) &\rightsquigarrow (\neg\neg X \wedge \neg Y) \vee (\neg Y \vee Z) \\
&\rightsquigarrow (X \wedge \neg Y) \vee (\neg Y \vee Z) \\
&\rightsquigarrow (X \vee \neg Y \vee Z) \wedge (\neg Y \vee \neg Y \vee Z)
\end{aligned}
$$

Efficient computation of the clausal form. Given a clause C, we write $L(C)$ for the set of literals occurring in it:

$$
L(X) = \{X\} \quad L(\neg X) = \{\neg X\} \quad L(C \vee D) = L(C) \cup L(D) \quad L(\bot) = \emptyset
$$

A variable X occurs *positively* (resp. *negatively*) in A if we have $X \in L(C)$ (resp. $\neg X \in L(C)$). Up to equivalence, formulas satisfy the laws of commutative idempotent monoids with respect to $\vee$ and $\bot$:

$$
\begin{aligned}
(A \vee B) \vee C \Leftrightarrow A \vee (B \vee C) && \bot \vee A \Leftrightarrow A && B \vee A \Leftrightarrow A \vee B \\
&& A \vee \bot \Leftrightarrow A && A \vee A \Leftrightarrow A
\end{aligned}
$$

Because of this, a clause is characterized by the set of literals occurring in it, see appendix A.2:

Lemma 2.5.5.3. Given clauses C and D, if $L(C) = L(D)$ then $C \Leftrightarrow D$.

Similarly, a formula in clausal form is characterized by the set of clauses occurring in it. A formula in clausal form A can thus be encoded as a set of sets of literals:

$$
A = \{\{L_1^1, \dots, L_{k_1}^1\}, \dots, \{L_1^n, \dots, L_{k_n}^n\}\}
$$

Note that the empty set $\emptyset$ corresponds to the formula $\top$ whereas the set $\{\emptyset\}$ corresponds to the formula $\bot$. In practice, we can represent a formula as a list of lists of clauses (where the order or repetitions of the elements of the lists do not matter). Based on this, an algorithm for putting a formula in clausal form is provided in figure 2.6. A literal is encoded as a pair consisting of a variable and a boolean indicating whether it is negated or not (by convention, `false` means negated), a clause as a list of literals, and clausal form as a list of clauses. Given a formula A, the functions `pos` and `neg` compute the clausal form of A and $\neg A$, respectively. They are using the function `merge`, which, given two formulas in clausal form

$$
A = \{C_1, \dots, C_m\} \qquad \text{and} \qquad B = \{D_1, \dots, D_n\}
$$

computes the clausal form of $A \vee B$, which is

$$
A \vee B = \{C_i \cup D_j \mid 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n\}
$$

The notion of clausal form can be further improved as follows. We say that a formula is in *canonical clausal form* when

1. it is in clausal form,

```
type var = int

(** Formulas. *)
type t =
  | Var of var
  | And of t * t
  | Or  of t * t
  | Imp of t * t
  | Not of t
  | True | False

type literal = bool * var   (** Literal. *) (* false = negated *)
type clause  = literal list (** Clause. *)
type cnf     = clause list  (** Clausal formula. *)

let clausal a : cnf =
  let merge a b =
    List.flatten (List.map (fun c -> List.map (fun d -> c@d) b) a)
  in
  let rec pos = function
    | Var x       -> [[true, x]]
    | And (a, b) -> let a = pos a in let b = pos b in a@b
    | Or  (a, b) -> let a = pos a in let b = pos b in merge a b
    | Imp (a, b) -> let a = neg a in let b = pos b in merge a b
    | Not a       -> neg a
    | True        -> []
    | False       -> [[]]
  and neg = function
    | Var x       -> [[false, x]]
    | And (a, b) -> let a = neg a in let b = neg b in merge a b
    | Or  (a, b) -> let a = neg a in let b = neg b in a@b
    | Imp (a, b) -> let a = pos a in let b = neg b in a@b
    | Not a       -> pos a
    | True        -> [[]]
    | False       -> []
  in
  pos a
```

Figure 2.6: Rewriting a formula to a clausal form.

2. it does not contain twice the same clause or $\top$ (this is automatic if it is represented as a set of clauses),

3. no clause contains twice the same literal or $\bot$ (this is automatic if they are represented as sets of literals),

4. no clause contains both a literal and its negation.

For the last point, given a clause C containing both X and $\neg X$, the equivalences $\neg X \vee X \Leftrightarrow \top$ and $\top \vee A \Leftrightarrow \top$ imply that the whole clause is equivalent to $\top$ and can thus be removed from the formula. For instance, the clausal form computed in example 2.5.5.2 is not canonical because it does not satisfy the second point above.

Exercise 2.5.5.4. Modify the algorithm of figure 2.6 so that it computes canonical clausal forms.

De Morgan laws in intuitionistic logic. Let us insist once again on the fact that the de Morgan laws do not hold in intuitionistic logic. Namely, the following implications are intuitionistically true, but not their converse:

$$A \vee B \Rightarrow \neg(\neg A \wedge \neg B) \qquad \neg A \vee \neg B \Rightarrow \neg(A \wedge B)$$
$$A \wedge B \Rightarrow \neg(\neg A \vee \neg B) \qquad \neg A \vee B \Rightarrow A \Rightarrow B$$

However, the following equivalence does hold intuitionistically:

$$\neg A \wedge \neg B \Leftrightarrow \neg(A \vee B)$$

2.5.6 Boolean models. Classical natural deduction matches exactly the notion of truth one would get from usual boolean models. Let us detail this. We write $\mathbb{B} = \{0, 1\}$ for the set of *booleans*. A *valuation* ρ is a function $\mathcal{X} \to \mathbb{B}$, assigning booleans to variables. Such a valuation can be extended to a function $\overline{\rho} : \mathrm{Prop} \to \mathbb{B}$, from propositions to booleans, by induction over the propositions by

$$\overline{\rho}(X) = 1 \text{ iff } \rho(X) = 1$$
$$\overline{\rho}(A \Rightarrow B) = 1 \text{ iff } \overline{\rho}(A) = 0 \text{ or } \overline{\rho}(B) = 1$$
$$\overline{\rho}(A \wedge B) = 1 \text{ iff } \overline{\rho}(A) = 1 \text{ and } \overline{\rho}(B) = 1$$
$$\overline{\rho}(\top) = 1$$
$$\overline{\rho}(A \vee B) = 1 \text{ iff } \overline{\rho}(A) = 1 \text{ or } \overline{\rho}(B) = 1$$
$$\overline{\rho}(\bot) = 0$$

Given a formula A and a valuation ρ, we write $\vDash_\rho A$ whenever $\overline{\rho}(A) = 1$ and say that the formula A is *satisfied* in the valuation ρ. Given a context $\Gamma = A_1, \ldots, A_n$, we write $\Gamma \vDash_\rho A$ whenever $\vDash_\rho (\bigwedge_{i=1}^n A_i) \Rightarrow A$. Finally, we write $\Gamma \vDash A$ whenever $\Gamma \vDash_\rho A$ for every valuation ρ and, in this case, say that A is *valid* in the context Γ or that the sequent $\Gamma \vdash A$ is valid.

The system NK is correct in the sense that it only allows the derivation of valid sequents.

Theorem 2.5.6.1 (Soundness). If $\Gamma \vdash A$ is derivable then $\Gamma \vDash A$.

Proof. By induction on the proof of $\Gamma \vdash A$. □

Since NJ is a subsystem of NK, it thus also allows only the derivation of valid sequents. As simple as it may seem, the above theorem allows proving the consistency of intuitionistic and classical logic (which was already demonstrated in theorem 2.3.4.2 for intuitionistic logic):

Corollary 2.5.6.2. The system NK (and thus also NJ) is consistent.

Proof. Suppose that NK is not consistent. By lemma 2.3.4.1, we would have a proof of $\vdash \bot$. By theorem 2.5.6.1, we would have $\overline{\rho}(\bot) = 1$. But $\overline{\rho}(\bot) = 0$ by definition, contradiction. □

Conversely, we can show that the system NK is *complete*, meaning that if a sequent $\Gamma \vdash A$ is valid, i.e. we have $\Gamma \vDash A$, then it is derivable. As a particular case, we will have that if a formula A is valid then it is provable, i.e. $\vdash A$ is derivable. We first need the following lemmas.

Lemma 2.5.6.3. For any formulas A and B, variable X and valuation ρ, we have $\overline{\rho}(A[B/X]) = \overline{\rho'}(A)$, where $\rho'(X) = \overline{\rho}(B)$ and $\rho'(Y) = \rho(Y)$ for $X \neq Y$.

Proof. By induction on A. □

Lemma 2.5.6.4. For any formula A, the formula

$$((X \Rightarrow A[\top/X]) \wedge (\neg X \Rightarrow A[\bot/X])) \Rightarrow A$$

is derivable in NK.

Proof. For conciseness, we write

$$\delta_X A = (X \Rightarrow A[\top/X]) \wedge (\neg X \Rightarrow A[\bot/X])$$

We reason by induction on the formula A. If $A = X$ then

$$\delta_X X = (X \Rightarrow \top) \wedge (\neg X \Rightarrow \bot)$$

and we have

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{
        \cfrac{\overline{\delta_X X, \neg X \vdash (X \Rightarrow \top) \wedge (\neg X \Rightarrow \bot)} \ \text{(ax)}}{\delta_X X, \neg X \vdash \neg X \Rightarrow \bot} \ (\wedge_{\mathrm{E}}^{\mathrm{r}}) \qquad \overline{\delta_X X, \neg X \vdash \neg X} \ \text{(ax)}
      }{\delta_X X, \neg X \vdash \bot} \ (\Rightarrow_{\mathrm{E}})
    }{\delta_X X \vdash \neg\neg X} \ (\neg_{\mathrm{I}})
  }{\delta_X X \vdash X} \ (\neg\neg_{\mathrm{E}})
}{\vdash \delta_X X \Rightarrow X} \ (\Rightarrow_{\mathrm{I}})
$$

If $A = Y$ with $Y \neq X$, we have

$$\delta_X Y = (X \Rightarrow Y) \wedge (\neg X \Rightarrow Y)$$

and, using the fact that $X \vee \neg X$ is derivable,

$$
\cfrac{
  \delta_X Y \vdash X \vee \neg X \qquad
  \cfrac{
    \cfrac{\cfrac{\overline{\delta_X Y, X \vdash (X \Rightarrow Y) \wedge (\neg X \Rightarrow Y)} \ \text{(ax)}}{\delta_X Y, X \vdash X \Rightarrow Y} \ (\wedge_{\mathrm{E}}^{\mathrm{l}}) \qquad \overline{\delta_X Y, X \vdash X} \ \text{(ax)}}{\delta_X Y, X \vdash Y} \ (\Rightarrow_{\mathrm{E}}) \qquad \cfrac{\vdots}{\delta_X Y, \neg X \vdash Y}
  }{\delta_X Y \vdash Y}
}{} \ (\vee_{\mathrm{E}})
$$

Other cases are left to the reader. □

Theorem 2.5.6.5 (Completeness). If $\Gamma \vDash A$ holds then $\Gamma \vdash A$ is derivable.

Proof. We proceed by induction on the number of free variables of A. If $\mathrm{FV}(A) = \emptyset$ then we easily show that $\Gamma \vdash A$ by induction on A. Otherwise, pick a variable $X \in \mathrm{FV}(A)$. By lemma 2.5.6.3, the sequents $\Gamma, X \vdash A[\top/X]$ and $\Gamma, \neg X \vdash A[\bot/X]$ are valid, and thus derivable by induction hypothesis. Moreover, lemma 2.5.6.4 states that $\delta_X A \Rightarrow A$ is derivable. We thus have the derivation

$$
\cfrac{
\cfrac{\vdots}{\Gamma \vdash \delta_X A \Rightarrow A}
\qquad
\cfrac{
\cfrac{\cfrac{\vdots}{\Gamma, X \vdash A[\top/X]}}{\Gamma \vdash X \Rightarrow A[\top/X]} (\Rightarrow_{\mathrm{I}})
\qquad
\cfrac{\cfrac{\vdots}{\Gamma, \neg X \vdash A[\bot/X]}}{\Gamma \vdash \neg X \Rightarrow A[\bot/X]} (\Rightarrow_{\mathrm{I}})
}{\Gamma \vdash \delta_X A} (\wedge_{\mathrm{I}})
}{\Gamma \vdash A} (\Rightarrow_{\mathrm{E}})
$$

which allows us to conclude. $\qquad\square$

A detailed and formalized version of this proof can be found in [CKA15].

Of course, intuitionistic natural deduction is not complete with respect to boolean models since there are formulas, such as $\neg X \vee X$, which evaluate to true under any valuation but are not derivable (lemma 2.3.5.2). One way to understand this is that there are "not enough boolean models" in order to detect that such formulas are not valid. A natural question is thus: is there a way to generalize the notion of boolean model, so that intuitionistic natural deduction is complete with respect to this generalized notion of model, i.e. a formula which is valid in any such a model is necessarily intuitionistically provable? We will see in section 2.8 that such a notion of model exists: Kripke models.

2.5.7 DPLL. As an aside, we would like to present the usual algorithm to decide the satisfiability of boolean formulas, which is based on the previous observations. A propositional formula A is *satisfiable* when there exists a valuation ρ making it true, i.e. such that $\vDash_\rho A$. An efficient way to test whether this is the case or not is the *DPLL* algorithm, due to Davis, Putnam, Logemann and Loveland [DLL62]. The basic idea here is the one we have already seen in lemma 2.5.6.4: if the formula A is satisfiable by a valuation ρ then, given a variable X occurring in A, we have either $\rho(X) = 0$ or $\rho(X) = 1$ and we can test whether A is satisfiable in both cases recursively since this makes the number of variables decrease in the formula (we call this *splitting* on the variable X):

Lemma 2.5.7.1. Given a variable X, a formula A is satisfiable if and only if the formula $A[\bot/X]$ or $A[\top/X]$ is satisfiable.

Proof. If A is satisfiable, then there is a valuation ρ such that $\overline{\rho}(A) = 1$. If $\rho(X) = 0$ (resp. $\rho(X) = 1$) then, by lemma 2.5.6.3, $\overline{\rho}(A[\bot/X]) = \overline{\rho}(A) = 1$ (resp. $\overline{\rho}(A[\top/X]) = \overline{\rho}(A) = 1$) and therefore $A[\bot/X]$ (resp. $A[\top/X]$) is satisfiable. Conversely, if $A[\bot/X]$ (resp. $A[\top/X]$) is satisfiable by a valuation ρ then, writing ρ' for the valuation such that $\rho'(X) = 0$ (resp. $\rho'(X) = 1$) and $\rho'(Y) = \rho(Y)$ for $Y \neq X$, by lemma 2.5.6.3 we have $\overline{\rho}(A[\bot/X]) = \overline{\rho}'(A) = 1$ (resp. $\overline{\rho}(A[\top/X]) = \overline{\rho}'(A) = 1$). $\qquad\square$

In the base case, the formula A has no variable and it thus evaluates to the same value in any environment, and we can easily compute this value: it is satisfiable if and only if this value is true. This directly leads to a very simple implementation of a satisfiability algorithm, see figure 2.7: the function `subst` computes the substitution of a formula into another one, the function `var` finds a free variable, and finally the function `sat` tests the satisfiability of a formula.

As is, this algorithm is not very efficient: some subformulas get evaluated many times during the search. It can however be much improved by using formulas in canonical clausal form, as described in proposition 2.5.5.1. First, substitution can be implemented on those as follows:

Lemma 2.5.7.2. Given a canonical clausal formula A and a variable X, a canonical clausal formula for $A[\top/X]$ (resp. $A[\bot/X]$) can be obtained from A by

- removing all clauses containing X (resp. $\neg X$),

- removing $\neg X$ (resp. X) from all remaining clauses.

The computation can be further improved by carefully choosing the variables we are going to split on first. A *unitary clause* in a clausal formula A is a clause containing exactly one literal L. If L is X (resp. $\neg X$) then, if we split on X, the branch $A[\bot/X]$ (resp. $A[\top/X]$) will fail. Therefore,

Lemma 2.5.7.3. Consider a clausal formula A containing a unitary clause which is a literal X (resp. $\neg X$). Then the formula A is satisfiable if and only if the formula $A[\top/X]$ (resp. $A[\bot/X]$) is.

A literal X (resp. $\neg X$) is *pure* in a clausal formula A if $\neg X$ (resp. X) does not occur in any clause of A: the variable X always occurs with the same polarity (positive or negative) in the formula.

Lemma 2.5.7.4. A clausal formula A containing a pure literal X (resp. $\neg X$) is satisfiable if and only if the formula $A[\top/X]$ (resp. $A[\bot/X]$) is satisfiable.

Another way to state the above lemma is that the clauses containing the pure literal can be removed from the formula without changing its satisfiability.

The DPLL algorithm exploits these optimizations in order to test the satisfiability of formula A:

1. it first tries to see if A is obviously satisfiable (if it is $\top$) or unsatisfiable (if it contains the clause $\bot$),

2. otherwise it tries to find a unitary clause and apply lemma 2.5.7.3,

3. otherwise it tries to find a pure clause and apply lemma 2.5.7.4,

4. otherwise it splits on an arbitrary variable by lemma 2.5.7.1.

For the last step, various heuristics have been proposed for choosing the splitting variable such as MOM (a variable with Maximal number of Occurrences in the clauses of Minimum size) or Jeroslow-Wang (a variable with maximum $J(X) = \sum_C 2^{-|C|}$ where C ranges over clauses containing X and $|C|$ is the number of literals), and so on.

A concrete implementation is provided in figure 2.8. The function `sub` implements substitution as described in lemma 2.5.7.2, the function `unit` finds a unitary clause (or raises `Not_found` if there is none), the function `pure` finds a

```
(** Formulas. *)
type t =
  | Var of int
  | And of t * t
  | Or  of t * t
  | Not of t
  | True | False

(** Substitute a variable by a formula in a formula. *)
let rec subst x c = function
  | Var y      -> if x = y then c else Var y
  | And (a, b) -> And (subst x c a, subst x c b)
  | Or  (a, b) -> Or  (subst x c a, subst x c b)
  | Not a      -> Not (subst x c a)
  | True -> True | False -> False

(** Find a free variable in a formula. *)
let var a =
  let exception Found of int in
  let rec aux = function
    | Var x -> raise (Found x)
    | And (a, b) | Or (a, b) -> aux a; aux b
    | Not a -> aux a
    | True | False -> ()
  in
  try aux a; raise Not_found
  with Found x -> x

(** Evaluate a closed formula. *)
let rec eval = function
  | Var _      -> assert false
  | And (a, b) -> eval a && eval b
  | Or  (a, b) -> eval a || eval b
  | Not a      -> not (eval a)
  | True -> true | False -> false

(** Simple-minded satisfiability. *)
let rec sat a =
  try
    let x = var a in
    sat (subst x True a) || sat (subst x False a)
  with Not_found -> eval a
```

Figure 2.7: Naive implementation of the satisfiability algorithm.

```
type var     = int          (** Variable. *)
type literal = bool * var    (** Literal. *) (* false means negated *)
type clause  = literal list  (** Clause. *)
type cnf     = clause list   (** Clausal formula. *)

(** Substitution a[v/x]. *)
let subst (a:cnf) (v:bool) (x:var) : cnf =
  let a = List.filter (fun c -> not (List.mem (v,x) c)) a in
  List.map (fun c -> List.filter (fun l -> l <> (not v, x)) c) a

(** Find a unitary clause. *)
let rec unit : cnf -> literal = function
  | [n,x]::a -> n,x
  | _::a     -> unit a
  | []       -> raise Not_found

(** Find a pure literal in a clausal formula. *)
let pure (a : cnf) : literal =
  let rec clause vars = function
    | []       -> vars
    | (n,x)::c ->
      try
        match List.assoc x vars with
        | Some n' ->
          if n' = n then clause vars c else
            let vars = List.filter (fun (y,_) -> y <> x) vars in
            clause ((x,None)::vars) c
        | None    -> clause vars c
      with Not_found -> clause ((x,Some n)::vars) c
  in
  let vars = List.fold_left clause [] a in
  let x, n = List.find (function (x,Some s) -> true | _ -> false) vars in
  Option.get n, x

(** DPLL procedure. *)
let rec dpll a =
  if a = [] then true
  else if List.mem [] a then false
  else
    try let n,x = unit a in dpll (subst a n x)
    with Not_found ->
    try let n,x = pure a in dpll (subst a n x)
    with Not_found ->
      let x = snd (List.hd (List.hd a)) in
      dpll (subst a false x) || dpll (subst a true x)
```

Figure 2.8: Implementation of the DPLL algorithm.

pure literal (or raises `Not_found`) and finally the function `dpll` implements the above algorithm. The function `pure` uses an auxiliary list `vars` of pairs X,b where X is a variable and b is either `Some true` or `Some false` if the variable X occurs only positively or negatively, or `None` if it occurs both positively and negatively.

2.5.8 Resolution. The resolution procedure is a generalization of the previous DPLL algorithm which was introduced by Davis and Putnam [DP60]. It is not the most efficient algorithm, but one of the main interesting points about it is that it generalizes well to first-order logic, see section 5.4.6. It stems from the following observation.

Lemma 2.5.8.1 (Correctness). Suppose given two clauses of the form $C \vee X$ and $\neg X \vee D$, containing a variable X and its negation $\neg X$, respectively. Then the formula $C \vee D$ is a consequence of them.

Proof. Given a valuation ρ such that $\overline{\rho}(C \vee X) = \overline{\rho}(\neg X \vee D) = 1$,

   – if $\overline{\rho}(X) = 1$ then necessarily $\overline{\rho}(D) = 1$ and thus $\overline{\rho}(C \vee D) = 1$,

   – if $\overline{\rho}(X) = 0$ then necessarily $\overline{\rho}(C) = 1$ and thus $\overline{\rho}(C \vee D) = 1$.   □

From a logical point of view, this deduction corresponds to the following *resolution rule*:

$$\frac{\Gamma \vdash C \vee X \qquad \Gamma \vdash \neg X \vee D}{\Gamma \vdash C \vee D} \text{ (res)}$$

In the following, we implicitly consider formulas up to commutativity of disjunction, i.e. identify the formulas $A \vee B$ and $B \vee A$, so that the above rule also applies to clauses containing X and its negation:

$$\frac{\Gamma \vdash C_1 \vee X \vee C_2 \qquad \Gamma \vdash D_1 \vee \neg X \vee D_2}{\Gamma \vdash C_1 \vee C_2 \vee D_1 \vee D_2} \text{ (res)}$$

The previous lemma can be reformulated in classical logic as follows:

Lemma 2.5.8.2. The resolution rule is admissible in classical natural deduction.

Proof. We have

$$\frac{\dfrac{\dfrac{\Gamma \vdash C' \vee X}{\Gamma \vdash C', X}}{\Gamma \vdash C', X, D'} \text{ (wk}_\text{R}) \qquad \dfrac{\dfrac{\Gamma \vdash \neg X \vee D'}{\Gamma \vdash \neg X, D'}}{\Gamma \vdash C', \neg X, D'} \text{ (wk}_\text{R})}{\dfrac{\Gamma \vdash C', \bot, D'}{\Gamma \vdash C', D'} \text{ (}\bot_\text{R})} \text{ (}\neg_\text{E})$$

where the rule

$$\frac{\Gamma \vdash A \vee B}{\Gamma \vdash A, B}$$

is derivable by

$$\frac{\Gamma \vdash A \vee B \qquad \overline{\Gamma, A \vdash A, B} \text{ (ax)} \qquad \overline{\Gamma, B \vdash A, B} \text{ (ax)}}{\Gamma \vdash A, B} \text{ (}\vee_\text{E})$$

(in other words, the rule ($\vee_\text{I}$) is reversible).   □

Remark 2.5.8.3. If we recall that in classical logic implication can be defined as $A \Rightarrow B = \neg A \vee B$, the resolution rule simply corresponds to the transitivity of implication:

$$\frac{\Gamma \vdash \neg C \Rightarrow X \qquad \Gamma \vdash X \Rightarrow D}{\Gamma \vdash \neg C \Rightarrow D}$$

For simplicity, in the following a context Γ will be seen as a set of clauses (as opposed to a list of clauses, see section 2.2.10) and will also be interpreted as a clausal form (the conjunction of its clauses, see section 2.5.5). We will always implicitly suppose that it is canonical (see section 2.5.5): a clause cannot contain the same literal twice or a literal and its negation. Previous lemmas entail that we can prove the sequent $\Gamma \vdash \bot$ using axiom and resolution rules only when Γ is not satisfiable: otherwise, $\bot$ would also be satisfiable, which it is not by definition. We are going to show in theorem 2.5.8.7 that this observation admits a converse.

Resolvent. Given clauses $C \vee X$ and $\neg X \vee D$, the clause $C \vee D$ does not contain the variable X, which gives us the idea of using resolution to remove the variable X from a set of formulas by performing all the possible deductions we can. Suppose given a set Γ of clauses and X a variable. We write

$$\Gamma_X = \{C \mid C \vee X \in \Gamma\} \qquad \Gamma_{\neg X} = \{D \mid \neg X \vee D \in \Gamma\}$$

and Γ' for the set of clauses in Γ which contain neither X nor $\neg X$. We supposed that the clauses are in canonical form, so that we have the following partition of Γ:

$$\Gamma = \Gamma' \uplus \{C \vee X \mid C \in \Gamma_X\} \uplus \{\neg X \vee D \mid D \in \Gamma_{\neg X}\}$$

The *resolvent* $\Gamma \setminus X$ of Γ with respect to X is

$$\Gamma \setminus X = \Gamma' \cup \{C \vee D \mid C \in \Gamma_X, D \in \Gamma_{\neg X}\}$$

Remark 2.5.8.4. As defined above, the resolvent might contain clauses not in canonical form, even if C and D are. In order to keep this invariant, we should remove all clauses of the form $C \vee D$ such that C contains a literal and D its negation, which we will implicitly do; in clauses, we should also remove duplicate literals.

As indicated above, computing the resolvent reduces the number of free variables of Γ:

Lemma 2.5.8.5. Given Γ in clausal form and a variable X, we have

$$\mathrm{FV}(\Gamma \setminus X) = \mathrm{FV}(\Gamma) \setminus \{X\}$$

Its main interest lies in the fact that it preserves satisfiability:

Lemma 2.5.8.6. Given a clausal form Γ and a variable X, Γ is satisfiable if and only if $\Gamma \setminus X$ is satisfiable.

Proof. The left-to-right implication follows from the correctness of the resolution rule (lemma 2.5.8.1). For the right-to-left implication, suppose that $\Gamma \setminus X$ is satisfied under a valuation ρ. We are going to show that Γ is satisfied under either ρ_0 or ρ_1, where ρ_i is defined, for $i = 0$ or $i = 1$, by $\rho_i(X) = i$ and $\rho_i(Y) = \rho(Y)$ whenever $Y \neq X$. We distinguish two cases.

– Suppose that we have $\rho(C') = 1$ for every clause $C = C' \vee X$ in Γ_X. Then we can take $i = 0$. Namely, given a clause $C \in \Gamma = \Gamma' \uplus \Gamma_X \uplus \Gamma_{\neg X}$:

- if $C \in \Gamma'$ then $\rho_0(C) = \rho(C) = 1$ because C does not contain the literal X,

- if $C \in \Gamma_X$ then $C = C' \vee X$ and $\rho_0(C) = 1$ because, by hypothesis, $\rho_0(C') = \rho(C') = 1$,

- if $C \in \Gamma_{\neg X}$ then $C = C' \vee \neg X$ and $\rho_0(C) = 1$ because $\rho_0(\neg X) = 1$ since $\rho_0(X) = 0$ by definition of ρ_0.

– Otherwise, there exists a clause $C = C' \vee X \in \Gamma_X$ such that $\rho(C') = 0$. Then we can take $i = 1$. Namely, given a clause $D \in \Gamma = \Gamma' \uplus \Gamma_X \uplus \Gamma_{\neg X}$:

- if $D \in \Gamma'$ then $\rho_1(D) = \rho(D) = 1$ because D does not contain the literal X,

- if $D \in \Gamma_X$ then $D = D' \vee X$ and $\rho_1(D) = 1$ because $\rho_1(X) = 1$,

- if $D \in \Gamma_{\neg X}$ then $D = D' \vee \neg X$ and $\rho(C' \vee D') = 1$ by hypothesis, thus $\rho_1(D') = \rho(D') = 1$ because $\rho(C') = 0$, thus $\rho_1(D) = \rho_1(D' \vee X) = 1$. $\square$

The previous lemma implies that resolution is *refutation complete* in the sense that it can always be used to show that a set of clauses cannot be satisfied (by whichever valuation):

Theorem 2.5.8.7 (Refutation completeness). A set Γ of clauses is unsatisfiable if and only if $\Gamma \vdash \bot$ can be proved using the axiom and resolution rules only.

Proof. Writing $\mathrm{FV}(\Gamma) = \{X_1, \ldots, X_n\}$ for the free variables of Γ, define the sequence of sets of clauses $\Gamma_{0 \leqslant i \leqslant n}$ by $\Gamma_0 = \Gamma$ and $\Gamma_{i+1} = \Gamma_i \setminus X_i$:

- the clauses of Γ_0 can be deduced from those of Γ using the axiom rule,

- the clauses of Γ_{i+1} can be deduced from those in Γ_i using the resolution rule.

Lemma 2.5.8.6 ensures that Γ_i is satisfiable if and only if Γ_{i+1} is satisfiable, and thus, by induction, Γ_0 is satisfiable if and only if Γ_n is satisfiable. Moreover, by lemma 2.5.8.5, we have $\mathrm{FV}(\Gamma_n) = \emptyset$, thus $\Gamma_n = \emptyset$ or $\Gamma_n = \{\bot\}$, and therefore Γ_n is unsatisfiable if and only if $\Gamma_n = \{\bot\}$. Finally, Γ is unsatisfiable if and only if $\Gamma_n = \{\bot\}$, i.e. $\bot$ can be deduced from Γ using axiom and resolution rules. $\square$

Completeness. Resolution is not complete: given a context Γ, there are clauses that can be deduced which cannot using resolution only. For instance, from $\Gamma = X$ we cannot deduce $X \vee Y$ using resolution only. However, resolution can be used in order to decide whether a formula A is a consequence of a context Γ, in the following way:

Lemma 2.5.8.8. A formula A is a consequence of a context Γ if and only if $\Gamma \cup \{\neg A\}$ is unsatisfiable.

Proof. Given a clausal form Γ, we have $\Gamma \Rightarrow A$ equivalent to $\neg\neg(\Gamma \Rightarrow A)$ equivalent to $\neg(\Gamma \wedge \neg A)$, i.e. $\Gamma \cup \{\neg A\}$ not satisfiable. $\square$

This lemma is the usual way we use resolution.

Example 2.5.8.9. We can show that given

$$X \Rightarrow Y \qquad\qquad Y \Rightarrow Z \qquad\qquad X$$

we can deduce Z. Rewriting those in normal form and using the previous lemma, this amounts to showing that Γ consisting of

$$\neg X \vee Y \qquad\qquad \neg Y \vee Z \qquad\qquad X \qquad\qquad \neg Z$$

is not satisfiable. Indeed, we have

$$
\cfrac{
\cfrac{
\cfrac{\cfrac{}{\Gamma \vdash \neg X \vee Y}\,(\text{ax}) \qquad \cfrac{}{\Gamma \vdash \neg Y \vee Z}\,(\text{ax})}{\Gamma \vdash \neg X \vee Z}\,(\text{res}) \qquad \cfrac{}{\Gamma \vdash X}\,(\text{ax})
}{\Gamma \vdash Z}\,(\text{res}) \qquad \cfrac{}{\Gamma \vdash \neg Z}\,(\text{ax})
}{\Gamma \vdash \bot}\,(\text{res})
$$

Implementation. We implement clausal forms using lists as in section 2.5.7. Using this representation, the resolvent of a clausal form Γ (written g) with respect to a variable X (written x) can be computed using the following function:

```
let resolve x g =
  let gx  = List.filter (List.mem (true ,x)) g   in
  let gx  = List.map (List.remove (true ,x)) gx  in
  let gx' = List.filter (List.mem (false,x)) g   in
  let gx' = List.map (List.remove (false,x)) gx' in
  let g'  = List.filter (List.for_all (fun (_,y) -> y <> x)) g in
  let disjunction c d =
    let union c d =
      List.fold_left
        (fun d l -> if List.mem l d then d else l::d)
        d c
    in
    if c = [] && d = [] then raise False
    else
    if List.exists (fun (n,x) -> List.mem (not n,x) d) c then None
    else Some (union c d)
  in
  g'@(List.filter_map_pairs disjunction gx gx')
```

Here, g' is Γ' and gx is Γ_X and gx' is $\Gamma_{\neg X}$ and we return the resolvent computed following the definition

$$\Gamma \setminus X = \Gamma' \cup \{C \vee D \mid C \in \Gamma_X, D \in \Gamma_{\neg X}\}$$

The function List.filter_map_pairs, which is of type

```
('a -> 'b -> 'c option) -> 'a list -> 'b list -> 'c list
```

takes a function and two lists as arguments, applies the functions to every pair of elements of one list and the other, and returns the list of results which are

not `None`. It is used to compute the clauses $C \vee D$ in the definition of $\Gamma \setminus X$. The disjunction is computed by `disjunction`, with some subtleties. Firstly, as noted in remark 2.5.8.4, we should be careful in order to produce formulas in canonical form:

– a disjunction $C \vee D$ containing both a literal and its negation should not be added,

– in a disjunction $C \vee D$, if a literal occurs twice (once in C and once D), we should only keep one instance.

Secondly, since we want to detect as early as possible when $\bot$ can be deduced, we raise an exception `False` when we find one. We can then see whether a clausal form Γ is inconsistent by repeatedly eliminating free variables using resolution. We use the auxiliary function `free_var` in order to find a free variable (it raises `Not_found` if there is none), its implementation being left to the reader. By theorem 2.5.8.7, if Γ is inconsistent then $\bot$ will be produced during the process (in which case the exception `False` is raised); otherwise the free variables will be exhausted (in which case the exception `Not_found` is raised). This can thus be computed with the following function:

```
let rec inconsistent g =
  try inconsistent (resolve (free_var g) g)
  with
  | False     -> true
  | Not_found -> false
```

We can then decide whether a clause is a consequence of a set of other clauses by applying lemma 2.5.8.8:

```
let prove g c =
  inconsistent ((neg c)::g)
```

As an application, we can prove example 2.5.8.9 with

```
let () =
  let g = [
    [false,0;true,1];
    [false,1;true,2];
    [true,0]
  ] in
  let c = [true,2] in
  assert (prove g c)
```

2.5.9 Double-negation translation. We have seen in section 2.3.5 that some formulas are not provable in intuitionistic logic, whereas they are valid in classical logic, a typical example being excluded middle $\neg A \vee A$. But can we really prove less in intuitionistic logic than in classical logic? A starting observation is that, even though $\neg A \vee A$ is not provable, its double negation $\neg\neg(\neg A \vee A)$

becomes provable, as first seen on page 63:

$$
\cfrac{
\cfrac{\neg(\neg A \vee A) \vdash \neg(\neg A \vee A)}{} \ (ax)
\qquad
\cfrac{
\cfrac{
\cfrac{
\cfrac{
\cfrac{\neg(\neg A \vee A), A \vdash \neg(\neg A \vee A)}{} \ (ax)
\qquad
\cfrac{\cfrac{\neg(\neg A \vee A), A \vdash A}{} \ (ax)}{\neg(\neg A \vee A), A \vdash \neg A \vee A} \ (\vee_{\mathrm{I}}^{\mathrm{r}})
}{\neg(\neg A \vee A), A \vdash \bot} \ (\neg_{\mathrm{E}})
}{\neg(\neg A \vee A) \vdash \neg A} \ (\neg_{\mathrm{I}})
}{\neg(\neg A \vee A) \vdash \neg A \vee A} \ (\vee_{\mathrm{I}}^{\mathrm{l}})
}{\neg(\neg A \vee A) \vdash \bot} \ (\neg_{\mathrm{E}})
}{\vdash \neg\neg(\neg A \vee A)} \ (\neg_{\mathrm{I}})
$$

One of the main ingredients behind this proof is that having $\neg(\neg A \vee A)$ as hypothesis in a context Γ allows to discard the current proof goal B and go back to proving $\neg A \vee A$:

$$
\cfrac{
\cfrac{
\cfrac{\Gamma \vdash \neg(\neg A \vee A)}{} \ (ax)
\qquad
\cfrac{\vdots}{\Gamma \vdash \neg A \vee A}
}{\Gamma \vdash \bot} \ (\neg_{\mathrm{E}})
}{\Gamma \vdash B} \ (\bot_{\mathrm{E}})
$$

How is this better than proving $\neg A \vee A$ directly? The fact that, during the proof, we can reset our proof goal to $\neg A \vee A$! We thus start by proving $\neg A \vee A$ by proving $\neg A$, which requires proving $\bot$ from A. At this point, we change our mind and start again the proof of $\neg A \vee A$, but this time we prove A, which we can because we gained this information from the previously "aborted" proof. A more detailed explanation of this kind of behavior was already developed in section 2.5.2. This actually generalizes to any formula, by a result due to Glivenko [Gli29]. Given a context Γ, we write $\neg\neg\Gamma$ for the context obtained from Γ by double-negating every formula.

Theorem 2.5.9.1 (Glivenko's theorem). Given a context Γ and propositional formula A, the sequent $\Gamma \vdash A$ is provable in classical logic if and only if the sequent $\neg\neg\Gamma \vdash \neg\neg A$ is provable in intuitionistic logic.

This result allows us to relate the consistency of classical and intuitionistic logic in the following way.

Theorem 2.5.9.2. Intuitionistic logic is consistent if and only if classical logic is consistent.

Proof. Suppose that intuitionistic logic is inconsistent: there is an intuitionistic proof of $\bot$. This proof is also a valid classical proof and thus classical logic is inconsistent. Conversely, suppose that classical logic is inconsistent. There is a classical proof of $\bot$ and thus, by theorem 2.5.9.1, an intuitionistic proof π of $\neg\neg\bot$. However, the implication $\neg\neg\bot \Rightarrow \bot$ holds intuitionistically:

$$
\cfrac{
\cfrac{
\cfrac{\neg\neg\bot \vdash \neg\neg\bot}{} \ (ax)
\qquad
\cfrac{\cfrac{\neg\neg\bot, \bot \vdash \bot}{} \ (ax)}{\neg\neg\bot \vdash \neg\bot} \ (\neg_{\mathrm{I}})
}{\neg\neg\bot \vdash \bot} \ (\neg_{\mathrm{E}})
}{\vdash \neg\neg\bot \Rightarrow \bot} \ (\Rightarrow_{\mathrm{I}})
$$

We thus have an intuitionistic proof of $\bot$:

$$
\cfrac{
\cfrac{\vdots}{\vdash \neg\neg\bot \Rightarrow \bot} \qquad \cfrac{\pi}{\vdash \neg\neg\bot}
}{\vdash \bot} \ (\Rightarrow_{\mathrm{E}})
$$

and intuitionistic logic is inconsistent. $\qquad\square$

Remark 2.5.9.3. The theorem 2.5.9.1 does not generalize as is to first-order logic, but some other translations of classical formulas into intuitionistic logic do. The most brutal one is due to Kolmogorov and consists in adding $\neg\neg$ in front of every subformula. Interestingly, it corresponds to the call-by-name continuation-passing style translation of functional programming languages. A more economical translation is due to Gödel, transforming a formula A into the formula A^* defined by induction:

$$
\begin{aligned}
X^* &= X \\
(A \wedge B)^* &= A^* \wedge B^* & (A \vee B)^* &= \neg(\neg A^* \wedge \neg B^*) \\
\top^* &= \top & \bot^* &= \bot \\
(A \Rightarrow B)^* &= A^* \Rightarrow B^* & (\neg A)^* &= \neg A^*
\end{aligned}
$$

Finally, one can wonder if, by adding four negations to a formula, we could gain even more proof power, but this is not the case: the process stabilizes after the first iteration.

Lemma 2.5.9.4. For every natural number $n > 0$, we have $\neg^{n+2} A \Leftrightarrow \neg^n A$.

Proof. The implication $A \Rightarrow \neg\neg A$ is intuitionistically provable, as already shown in example 2.2.5.3, as well as the implication $\neg\neg\neg A \Rightarrow \neg A$:

$$
\cfrac{
\cfrac{}{\neg\neg\neg A, A \vdash \neg\neg\neg A}\ (\mathrm{ax}) \qquad
\cfrac{
\cfrac{
\cfrac{\overline{\neg\neg\neg A, A, \neg A \vdash \neg A}\ (\mathrm{ax}) \qquad \overline{\neg\neg\neg A, A, \neg A \vdash A}\ (\mathrm{ax})}{\neg\neg\neg A, A, \neg A \vdash \bot}\ (\neg_{\mathrm{E}})
}{\neg\neg\neg A, A \vdash \neg\neg A}\ (\neg_{\mathrm{I}})
}{\neg\neg\neg A, A \vdash \bot}\ (\neg_{\mathrm{E}})
}{
\cfrac{\cfrac{\neg\neg\neg A, A \vdash \bot}{\neg\neg\neg A \vdash \neg A}\ (\neg_{\mathrm{I}})}{\vdash \neg\neg\neg A \Rightarrow \neg A}\ (\Rightarrow_{\mathrm{I}})
}
$$

We conclude by induction on n. $\qquad\square$

In particular, $\neg\neg\neg\neg A \Leftrightarrow \neg\neg A$, so that we gain nothing by performing the double-negation translation twice.

2.5.10 Intermediate logics. Once again, classical logic is obtained by adding the excluded middle $\neg A \vee A$ (or any of the equivalent axioms, see theorem 2.5.1.1) to intuitionistic logic, so that new formulas are provable. A natural question is: are there *intermediate logics* between intuitionistic and classical? This means: can we add axioms to intuitionistic logic so that we get strictly more than intuitionistic logic, but strictly less than classical logic? In more details: are there families of formulas, which are provable classically but not intuitionistically,

such that, by adding those as axioms to intuitionistic logic we obtain a logic in which some classical formulas are not provable?

The answer is yes. A typical such family of axioms is the *weak excluded middle*:

$$\neg\neg A \vee \neg A$$

Namely, the formula $\neg\neg X \vee \neg X$ is not provable in intuitionistic logic (see lemma 2.3.5.3), so that assuming the weak excluded middle (for every formula A) allows proving new formulas. However, the formula $\neg X \vee X$ does not follow from the weak excluded middle (example 2.8.1.4). There are many other possible families of axioms giving rise to intermediate logics such as

- *linearity* (or *Gödel-Dummett*) axiom [God32, Dum59]:

$$(A \Rightarrow B) \vee (B \Rightarrow A)$$

- Kreisel-Putnam axiom [KP57]:

$$(\neg A \Rightarrow (B \vee C)) \Rightarrow ((\neg A \Rightarrow B) \vee (\neg A \Rightarrow C))$$

- Scott's axiom [KP57]:

$$((\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)) \Rightarrow (\neg\neg A \vee \neg A)$$

- Smetanich's axiom [WZ07]:

$$(\neg B \Rightarrow A) \Rightarrow (((A \Rightarrow B) \Rightarrow A) \Rightarrow A)$$

- and many more [DMJ16].

Exercise 2.5.10.1. Show that the above linearity principle

$$(A \Rightarrow B) \vee (B \Rightarrow A)$$

is equivalent to the following global choice principle for disjunctions

$$(A \Rightarrow B \vee C) \Rightarrow (A \Rightarrow B) \vee (A \Rightarrow C)$$

2.6 Sequent calculus

Natural deduction is "natural" in the sense that it allows for a precise correspondence between logic and computation, see chapter 4. However, it has some flaws. From an aesthetic point of view, the rules for $\wedge$ and $\vee$ are not entirely dual, contrarily to what one would expect: if they were the same, we could think of reducing the work during proofs or implementations by handling them in the same way. More annoyingly, proof search is quite difficult. Namely, suppose that we are trying to prove

$$A \vee B \vdash B \vee A$$

The proof cannot begin with an introduction rule because we have no hope of filling the dots:

$$\frac{\begin{array}{c}\vdots\\\hline A \vee B \vdash B\end{array}}{A \vee B \vdash B \vee A}(\vee^{\mathrm{l}}_{\mathrm{I}}) \qquad\qquad \frac{\begin{array}{c}\vdots\\\hline A \vee B \vdash A\end{array}}{A \vee B \vdash B \vee A}(\vee^{\mathrm{r}}_{\mathrm{I}})$$

This means that we have to use another rule such as $(\vee_{\mathrm{E}})$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}(\vee_{\mathrm{E}})$$

which requires us to come up with a formula $A \vee B$ which is not directly indicated in the conclusion $\Gamma \vdash C$ and it is not clear how to automatically generate such formulas. Starting in this way, the proof can be ended as in example 2.2.5.2.

In order to overcome this problem, Gentzen has invented *sequent calculus*, which is another presentation of logic. In natural deduction, all rules operate on the formula on the right of $\vdash$ and there are introduction and elimination rules. In sequent calculus, there are only introduction rules, but those can operate either on formulas on the left or on the right of $\vdash$. This results in a highly symmetrical calculus.

2.6.1 Sequents. In sequent calculus, sequents are of the form

$$\Gamma \vdash \Delta$$

where Γ and Δ are contexts: the intuition is that we have the conjunction of formulas in Γ as hypothesis, from which we can deduce the disjunction of formulas in Δ.

2.6.2 Rules. In all the systems we consider, unless otherwise stated, we always suppose that we can permute, duplicate and erase formulas in context, i.e. that the structural rules of figure 2.9 are always present. The additional rules for sequent calculus are shown in figure 2.10 and the resulting system is called LK. In sequent calculus, as opposed to natural deduction, the symmetry between disjunction and conjunction has been restored: except for the axiom and cut, all rules come in a left and right flavor. Although the presentation is quite different, the provability power of this system is the same as the one for classical natural deduction presented in section 2.5:

Theorem 2.6.2.1. A sequent $\Gamma \vdash \Delta$ is provable in NK (figure 2.5) if and only if it is provable in LK (figure 2.10).

Proof. The idea is that, by induction, we can translate a proof in NK into a proof in LK, and back. The introduction rules in NK correspond to right rules in LK, the axiom rules match in both systems, the cut rule is admissible in NK (the proof is similar to the one for NJ in proposition 2.3.2.1), as well as various structural rules (shown as in section 2.2.7), so that we only have to show that the elimination rules of NK are admissible in LK and the left rules of LK are admissible in NK. We only handle the case of conjunction here:

$$\frac{\Gamma, B, A, \Gamma' \vdash \Delta}{\Gamma, A, B, \Gamma' \vdash \Delta} \;(\mathrm{xch_L}) \qquad\qquad \frac{\Gamma \vdash \Delta, B, A, \Delta'}{\Gamma \vdash \Delta, A, B, \Delta} \;(\mathrm{xch_R})$$

$$\frac{\Gamma, A, A, \Gamma' \vdash \Delta}{\Gamma, A, \Gamma' \vdash \Delta} \;(\mathrm{contr_L}) \qquad\qquad \frac{\Gamma \vdash \Delta, A, A, \Delta'}{\Gamma \vdash \Delta, A, \Delta'} \;(\mathrm{contr_R})$$

$$\frac{\Gamma, \Gamma' \vdash \Delta}{\Gamma, A, \Gamma' \vdash \Delta} \;(\mathrm{wk_L}) \qquad\qquad \frac{\Gamma \vdash \Delta, \Delta'}{\Gamma \vdash \Delta, A, \Delta'} \;(\mathrm{wk_R})$$

$$\frac{\Gamma, \top, \Gamma' \vdash \Delta}{\Gamma, \Gamma' \vdash \Delta} \;(\top_\mathrm{L}) \qquad\qquad \frac{\Gamma \vdash \Delta, \bot, \Delta'}{\Gamma \vdash \Delta, \Delta'} \;(\bot_\mathrm{R})$$

Figure 2.9: Structural rules for sequent calculus

$$\frac{}{\Gamma, A \vdash A, \Delta} \;(\mathrm{ax}) \qquad\qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta} \;(\mathrm{cut})$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \;(\wedge_\mathrm{L}) \qquad\qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \;(\wedge_\mathrm{R})$$

$$\frac{}{\Gamma \vdash \top, \Delta} \;(\top_\mathrm{R})$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \;(\vee_\mathrm{L}) \qquad\qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \;(\vee_\mathrm{R})$$

$$\frac{}{\Gamma, \bot \vdash \Delta} \;(\bot_\mathrm{L})$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta} \;(\Rightarrow_\mathrm{L}) \qquad\qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \;(\Rightarrow_\mathrm{R})$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \;(\neg_\mathrm{L}) \qquad\qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \;(\neg_\mathrm{R})$$

Figure 2.10: LK: rules of classical sequent calculus.

– the rule ($\wedge_{\mathrm{E}}^{\mathrm{l}}$) is admissible in LK:

$$
\cfrac{
\cfrac{
\cfrac{\vdots}{\Gamma \vdash A \wedge B, \Delta}
}{\Gamma \vdash A \wedge B, A, \Delta} \text{(wk}_\mathrm{R}\text{)}
\qquad
\cfrac{
\cfrac{}{\Gamma, A, B \vdash A, \Delta} \text{(ax)}
}{\Gamma, A \wedge B \vdash A, \Delta} \text{(}\wedge_\mathrm{L}\text{)}
}{\Gamma \vdash A, \Delta} \text{(cut)}
$$

and admissibility of ($\wedge_{\mathrm{E}}^{\mathrm{r}}$) is similar,

– the rule ($\wedge_{\mathrm{L}}$) is admissible in NK:

$$
\cfrac{
\cfrac{
\cfrac{}{\Gamma, A \wedge B \vdash A \wedge B, \Delta}\text{(ax)}
}{\Gamma, A \wedge B \vdash A, \Delta}\text{(}\wedge_\mathrm{E}^\mathrm{l}\text{)}
\quad
\cfrac{
\cfrac{
\cfrac{}{\Gamma, A \wedge B, A \vdash A \wedge B, \Delta}\text{(ax)}
}{\Gamma, A \wedge B, A \vdash B, \Delta}\text{(}\wedge_\mathrm{E}^\mathrm{r}\text{)}
\qquad
\cfrac{
\cfrac{\vdots}{\Gamma, A, B \vdash \Delta}
}{\Gamma, A \wedge B, A, B \vdash \Delta}\text{(wk}_\mathrm{R}\text{)}
}{\Gamma, A \wedge B, A \vdash \Delta}\text{(cut)}
}{\Gamma, A \wedge B \vdash \Delta}\text{(cut)}
$$

Other cases are similar. $\qquad\qquad\square$

Remark 2.6.2.2. As noted in [Gir89, chapter 5], the correspondence between proofs in NK and LK is not bijective. For instance, the two proofs in LK

$$
\cfrac{
\cfrac{
\cfrac{
\cfrac{
\cfrac{\cfrac{}{A, B \vdash A}\text{(ax)} \quad \cfrac{}{A, B \vdash B}\text{(ax)}}{A, B \vdash A \wedge B}\text{(}\wedge_\mathrm{R}\text{)}
}{A, B, B' \vdash A \wedge B}\text{(wk}_\mathrm{L}\text{)}
}{A, A', B, B' \vdash A \wedge B}\text{(wk}_\mathrm{L}\text{)}
}{A, A', B \wedge B' \vdash A \wedge B}\text{(}\wedge_\mathrm{L}\text{)}
}{A \wedge A', B \wedge B' \vdash A \wedge B}\text{(}\wedge_\mathrm{L}\text{)}
\qquad\qquad
\cfrac{
\cfrac{
\cfrac{
\cfrac{
\cfrac{\cfrac{}{A, B \vdash A}\text{(ax)} \quad \cfrac{}{A, B \vdash B}\text{(ax)}}{A, B \vdash A \wedge B}\text{(}\wedge_\mathrm{R}\text{)}
}{A, B, B' \vdash A \wedge B}\text{(wk}_\mathrm{L}\text{)}
}{A, A', B, B' \vdash A \wedge B}\text{(wk}_\mathrm{L}\text{)}
}{A \wedge A', B, B' \vdash A \wedge B}\text{(}\wedge_\mathrm{L}\text{)}
}{A \wedge A', B \wedge B' \vdash A \wedge B}\text{(}\wedge_\mathrm{L}\text{)}
$$

get mapped to the same proof in NK.

Mutiplicative presentation. An alternative presentation (called *multiplicative* presentation) of the rules is given in figure 2.11: instead of supposing that we have the same context, we can "merge" the contexts of the premises in the conclusion.

Single-sided presentation. By de Morgan laws, in classical logic we can suppose that only the variables are negated, and negated at most once, see section 2.5.5. For instance, $\neg(X \vee \neg Y)$ is equivalent to $\neg X \wedge Y$, which satisfies this property. Given a formula A of this form, we write A^* for a formula of this form equivalent to $\neg A$, which can be defined by induction:

$$
\begin{aligned}
X^* &= \neg X & (\neg X)^* &= X \\
(A \wedge B)^* &= A^* \vee B^* & (A \vee B)^* &= A^* \wedge B^* \\
\top^* &= \bot & \bot^* &= \top
\end{aligned}
$$

We omit implication here, since it can be defined as $A \Rightarrow B = A^* \vee B$. Now, it can be observed that proving a sequent of the form $\Gamma, A \vdash \Delta$ is essentially the same as proving the sequent $\Gamma \vdash A^*, \Delta$, except that all the rules get replaced by their opposites:

$$\frac{}{A \vdash A} \text{ (ax)} \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \text{ (cut)}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \text{ (}\wedge\text{L)} \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \wedge B, \Delta, \Delta'} \text{ (}\wedge\text{R)}$$

$$\frac{}{\Gamma \vdash \top, \Delta} \text{ (}\top\text{R)}$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \vee B \vdash \Delta, \Delta'} \text{ (}\vee\text{L)} \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \text{ (}\vee\text{R)}$$

$$\frac{}{\Gamma, \bot \vdash \Delta} \text{ (}\bot\text{L)}$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \Rightarrow B \vdash \Delta, \Delta'} \text{ (}\Rightarrow\text{L)} \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \text{ (}\Rightarrow\text{R)}$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \text{ (}\neg\text{L)} \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \text{ (}\neg\text{R)}$$

Figure 2.11: LK: rules of classical sequent calculus (multiplicative presentation).

Lemma 2.6.2.3. A sequent $\Gamma, A \vdash \Delta$ is provable in LK if and only if $\Gamma \vdash A^*, \Delta$ is.

For instance the proof on the left below corresponds to the proof on the right:

$$\frac{\dfrac{}{X \vdash X, \bot} \text{ (ax)}}{\dfrac{X, \neg X \vdash \bot}{X \wedge \neg X \vdash \bot} \text{ (}\wedge\text{L)}} \text{ (}\neg\text{L)} \qquad \frac{\dfrac{}{X \vdash X, \bot} \text{ (ax)}}{\dfrac{\vdash \neg X, X, \bot}{\vdash \neg X \vee X, \bot} \text{ (}\vee\text{R)}} \text{ (}\neg\text{R)}$$

Because of this, we can restrict the system to sequents of the form $\vdash \Delta$, which are called *single-sided*. All the rules preserve single-sidedness except for the axiom rule, which is easily modified in order to satisfy this property. With some extra care, we can even come up with a presentation which does not require any structural rules (those are admissible): the resulting presentation of the calculus is given in figure 2.12. If we do not want to consider only formulas where only variables can be negated, then the de Morgan laws can be added as the following explicit rules:

$$\frac{\vdash \neg A \vee \neg B, \Delta}{\vdash \neg(A \wedge B), \Delta} \qquad \frac{\vdash \neg A \wedge \neg B, \Delta}{\vdash \neg(A \vee B), \Delta} \qquad \frac{\vdash \bot, \Delta}{\vdash \neg\top, \Delta} \qquad \frac{\vdash \top, \Delta}{\vdash \neg\bot, \Delta} \qquad \frac{\vdash A, \Delta}{\vdash \neg\neg A, \Delta}$$

2.6.3 Intuitionistic rules. In order to obtain a sequent calculus adapted to intuitionistic logic, one should restrict the two-sided proof system to sequents of the form $\Gamma \vdash A$, i.e. those where the context on the right of $\vdash$ contains exactly one formula. We also have to take variants of rules such as ($\vee$R), which would

$$\frac{}{\vdash \Delta, A^*, \Delta', A, \Delta''} \text{ (ax)} \qquad\qquad \frac{}{\vdash \Delta, A, \Delta', A^*, \Delta''} \text{ (ax)}$$

$$\frac{\vdash \Delta, A, \Delta' \qquad \vdash \Delta, A^*, \Delta'}{\vdash \Delta, \Delta'} \text{ (cut)}$$

$$\frac{\vdash \Delta, A, \Delta' \qquad \vdash \Delta, B, \Delta'}{\vdash \Delta, A \wedge B, \Delta'} \text{ (} \wedge \text{)} \qquad\qquad \frac{\vdash \Delta, A, B, \Delta'}{\vdash \Delta, A \vee B, \Delta'} \text{ (} \vee \text{)}$$

$$\frac{}{\vdash \Delta, \top, \Delta'} \text{ (} \top \text{)} \qquad\qquad \frac{\vdash \Delta, \Delta'}{\vdash \Delta, \bot, \Delta'} \text{ (} \bot \text{)}$$

Figure 2.12: LK: single-sided presentation.

otherwise not maintain the invariant of having one formula on the right. With little more care, one can write rules which do not require adding structural rules (they are admissible): the resulting calculus is presented in figure 2.13. Note that in order for contraction to be admissible one has to keep $A \Rightarrow B$ in the context of the left premise. Similarly to theorem 2.6.2.1, one shows:

Theorem 2.6.3.1. A sequent $\Gamma \vdash A$ is provable in NJ if and only if it is provable in LJ.

2.6.4 Cut elimination. By a similar argument as in section 2.3.3, it can be shown:

Theorem 2.6.4.1. A sequent $\Gamma \vdash \Delta$ (resp. $\Gamma \vdash A$) is provable in LK (resp. LJ) if and only if it admits a proof without using the (cut) rule.

2.6.5 Proof search. From a proof-search point of view, sequent calculus is much more well-behaved than natural deduction since, with the exception of the cut rule, we do not have to come up with new formulas when searching for proofs:

Proposition 2.6.5.1. LK has the *subformula property*: apart from the (cut) rule, all the formulas occurring in the premise of a rule are subformulas of the formulas occurring in the conclusion.

Since, by theorem 2.6.4.1, we can look for proofs without cuts, this means that we never have to come up with a new formula during proof search! Moreover, there is no harm in applying a rule whenever it applies thanks to the following property:

Proposition 2.6.5.2. In LK, all the rules are reversible.

Implementation. We now implement proof search, which is most simple to do using the single-sided presentation, see figure 2.12. We describe formulas as

```
type t =
  | Var of bool * string (* false means negated variable *)
  | Imp of t * t
```

$$\dfrac{}{\Gamma, A, \Gamma' \vdash A} \text{ (ax)} \qquad\qquad \dfrac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B} \text{ (cut)}$$

$$\dfrac{\Gamma, A, B, \Gamma' \vdash C}{\Gamma, A \wedge B, \Gamma' \vdash C} \text{ ($\wedge_\text{L}$)} \qquad \dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \text{ ($\wedge_\text{R}$)}$$

$$\dfrac{}{\Gamma \vdash \top} \text{ ($\top_\text{R}$)}$$

$$\dfrac{\Gamma, A, \Gamma' \vdash C \quad \Gamma, B, \Gamma' \vdash C}{\Gamma, A \vee B, \Gamma' \vdash C} \text{ ($\vee_\text{L}$)} \qquad \dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \text{ ($\vee_\text{L}^\text{l}$)} \quad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \text{ ($\vee_\text{L}^\text{r}$)}$$

$$\dfrac{}{\Gamma, \bot, \Gamma' \vdash A} \text{ ($\bot_\text{L}$)}$$

$$\dfrac{\Gamma, A \Rightarrow B, \Gamma' \vdash A \quad \Gamma, B, \Gamma' \vdash C}{\Gamma, A \Rightarrow B, \Gamma' \vdash C} \text{ ($\Rightarrow_\text{L}$)} \qquad \dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \text{ ($\Rightarrow_\text{R}$)}$$

$$\dfrac{\Gamma, \neg A, \Gamma' \vdash A}{\Gamma, \neg A, \Gamma' \vdash \bot} \text{ ($\neg_\text{L}$)} \qquad\qquad \dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \text{ ($\neg_\text{R}$)}$$

Figure 2.13: Rules of intuitionistic sequent calculus (LJ).

```
| And of t * t
| Or  of t * t
| True | False
```

Using this representation, the negation of a formula can be computed with the function

```
let rec neg = function
  | Var (n, x) -> Var (not n, x)
  | Imp (a, b) -> And (a, neg b)
  | And (a, b) -> Or  (neg a, neg b)
  | Or  (a, b) -> And (neg a, neg b)
  | True       -> False
  | False      -> True
```

Finally, the following function implements proof search in LK:

```
let rec prove venv = function
  | [] -> false
  | a::env ->
    match a with
    | Var (n, x) ->
      List.mem (Var (not n, x)) venv ||
      prove ((Var (n, x))::venv) env
    | Imp (a, b) -> prove venv ((neg a)::b::env)
    | And (a, b) -> prove venv (a::env) && prove venv (b::env)
    | Or  (a, b) -> prove venv (a::b::env)
```

```
| True        -> true
| False       -> prove venv env
```

Since we are considering single sided sequents here, those can be encoded as lists of terms. The above function takes as argument a sequent $\Gamma' = $ `venv` and the sequent Γ to be proved. It picks a formula A in Γ and applies the rules of figure 2.12 on it, until A is split into a list of literals: once this is the case, those literals are put into the sequent Γ' (of already handled formulas). Initially, the context Γ' is empty, and we usually want to prove one formula A, so that we can define

```
let prove a = prove [] [a]
```

Proof search in intuitionistic logic. Proof search can be performed in LJ, but the situation is more subtle. First note that, similarly to the situation in LK (proposition 2.6.5.1), we have

Proposition 2.6.5.3. LJ has the subformula property.

As an immediate consequence, we deduce

Theorem 2.6.5.4. We can decide whether a sequent $\Gamma \vdash A$ is provable in LJ or not.

Proof. There is only a finite number of subformulas of $\Gamma \vdash A$. We can restrict to sequents where a formula occurs at most 3 times in the context [Gir11, section 4.2.2] and therefore there is a finite number of possible sequents formed with those subformulas. By testing all the possible rules, we can determine which of those are provable, and thus determine whether the initial sequent is provable. $\qquad\square$

The previous theorem is constructive, but the resulting algorithm is quite inefficient.

The problem of finding proofs is more delicate than for LK because not all the rules are reversible: $(\vee_L^l)$, $(\vee_L^r)$ and $(\Rightarrow_L)$ are not reversible. The rules $(\vee_L^l)$, $(\vee_L^r)$ are easy to handle when performing proof search: when trying to prove a formula $A \vee B$, we either try to prove A or to prove B. The rule $(\Rightarrow_L)$

$$\frac{\Gamma, A \Rightarrow B, \Gamma' \vdash A \qquad \Gamma, B, \Gamma' \vdash C}{\Gamma, A \Rightarrow B, \Gamma' \vdash C} \ (\Rightarrow_L)$$

is more difficult to handle. If we apply it naively, it can loop for the same reasons as in section 2.4.2:

$$\frac{\dfrac{\vdots}{\dfrac{\Gamma, A \Rightarrow B \vdash A}{\Gamma, A \Rightarrow B \vdash A}\ (\Rightarrow_L) \quad \dfrac{\vdots}{\Gamma, B \vdash B}}{\dfrac{\Gamma, A \Rightarrow B \vdash A}{}\ (\Rightarrow_L)} \quad \dfrac{\vdots}{\Gamma, B \vdash B}}{\Gamma, A \Rightarrow B \vdash A} \ (\Rightarrow_L)$$

Although we can detect loops by looking at whether we encounter the same sequent twice during the proof search, this is quite impractical. Also, since the rule $(\Rightarrow_L)$ is not reversible, the order in which we apply it during proof search

$$\frac{\Gamma, A, \Gamma' \vdash B \qquad X \in \Gamma, \Gamma'}{\Gamma, X \Rightarrow A, \Gamma' \vdash B} \ (\Rightarrow x)$$

$$\frac{\Gamma, B \Rightarrow C, \Gamma' \vdash A \Rightarrow B \qquad \Gamma, C \vdash D}{\Gamma, (A \Rightarrow B) \Rightarrow C, \Gamma' \vdash D} \ (\Rightarrow_\Rightarrow)$$

$$\frac{\Gamma, A \Rightarrow (B \Rightarrow C), \Gamma' \vdash D}{\Gamma, (A \wedge B) \Rightarrow C, \Gamma' \vdash D} \ (\Rightarrow_\wedge) \qquad \frac{\Gamma, A \Rightarrow C, B \Rightarrow C, \Gamma' \vdash D}{\Gamma, (A \vee B) \Rightarrow C, \Gamma' \vdash D} \ (\Rightarrow_\vee)$$

$$\frac{\Gamma, A, \Gamma' \vdash B}{\Gamma, \top \Rightarrow A, \Gamma' \vdash B} \ (\Rightarrow_\top) \qquad \frac{\Gamma, \Gamma' \vdash B}{\Gamma, \bot \Rightarrow A, \Gamma' \vdash B} \ (\Rightarrow_\bot)$$

Figure 2.14: Left implication rules in LJT.

is relevant, and we would like to minimize the number of times we have to backtrack.

The logic LJT was introduced by Dyckoff in order to overcome this problem [Dyc92]. It is obtained from LJ by replacing the $(\Rightarrow_L)$ rule with the six rules of figure 2.14, which allow proving sequents of the form

$$\Gamma, A \Rightarrow B, \Gamma' \vdash C$$

depending on the form of A.

Proposition 2.6.5.5. A sequent is provable in LJ if and only if it is provable in LJT.

The main interest of this variant is that proof search is always terminating (thus the T in LJT). Moreover, the rules $(\Rightarrow_\wedge)$, $(\Rightarrow_\vee)$, $(\Rightarrow_\top)$ and $(\Rightarrow_\bot)$ are reversible and can thus always be applied during proof search. Many variants of this idea have been explored, such as the SLJ calculus [GLW99].

A proof search procedure based on this sequent calculus can be implemented as follows. We describe terms as usual as

```
type t =
  | Var of string
  | Imp of t * t
  | And of t * t
  | Or  of t * t
  | True | False
```

The procedure which determines whether a formula is provable is then shown in figure 2.15. This procedure takes as argument two contexts Γ' and Γ (respectively called env' and env) and a formula A. Initially, the context Γ' is empty; it will be used to store the formulas of Γ which have already been "processed". The procedure first applies all the reversible right rules, then all the reversible left rules; a formula of Γ which does not give rise to a reversible left rule is put in Γ'. Once this is done, the procedure tries to apply the axiom rule, handles disjunctions by trying to apply either $(\vee_L^l)$ or $(\vee_L^r)$, and finally successively tries

```
let rec prove env' env a =
  match a with
  | True -> true
  | And (a, b) -> prove env' env a && prove env' env b
  | Imp (a, b) -> prove env' (a::env) b
  | _ -> match env with
  | b::env -> (match b with
      | And (b, c) -> prove env' (b::c::env) a
      | Or  (b, c) -> prove env' (b::env) a && prove env' (c::env) a
      | True       -> prove env' env a
      | False      -> true
      | Imp (And (b, c), d) ->
        prove env' ((Imp (b, Imp (c,d)))::env) a
      | Imp (Or  (b, c), d) ->
        prove env' ((Imp (b,d))::(Imp (c,d))::env) a
      | Imp (True , b)      ->
        prove env' (b::env) a
      | Imp (False, b)      ->
        prove env' env a
      | Var _ | Imp (Var _, _) | Imp (Imp (_,_),_) ->
        prove (b::env') env a
    )
  | [] ->
    match a with
    | Var _ when List.mem a env' -> true
    | Or (a, b) -> prove env' env a || prove env' env b
    | a ->
      List.exists
        (fun (b, env') ->
           match b with
           | Imp (Var x, b) when List.mem (Var x) env' ->
             prove env' [b] a
           | Imp (Imp (b, c), d) ->
             prove env' [Imp (c, d)] (Imp (b, c)) && prove env' [d] a
           | _ -> false
        ) (context_formulas env')

let prove a = prove [] [] a
```

Figure 2.15: Proof search in LJT

all the possible applications of the non-reversible rules $(\Rightarrow_X)$ and $(\Rightarrow_\Rightarrow)$. Here the function `context_formulas` returns, given a context Γ, the list of all the pairs consisting of a formula A and a context Γ', Γ'' such that $\Gamma = \Gamma', A, \Gamma''$, i.e. the context Γ where some formula A has been removed.

2.7 Hilbert calculus

The *Hilbert calculus* is another formalism, due to Hilbert [Hil22], which makes opposite "design choices" than previous formalisms (natural deduction and sequent calculus): it has lots of axioms and very few logical rules.

2.7.1 Proofs. In this formalism, sequents are of the form $\Gamma \vdash A$, with Γ a context and A a formula, and are deduced according to the following two rules only:

$$\frac{}{\Gamma, A, \Gamma' \vdash A} \; (\mathrm{ax}) \qquad\qquad \frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B} \; (\Rightarrow_\mathrm{E})$$

respectively called *axiom* and *modus ponens*. Of course, there is very little that we can deduce with only these two rules. The other necessary logical principles are added in the form of axiom schemes, which can be assumed at any time during the proofs. In the case of the implicational fragment (implication is the only connective with which the formulas are built), those are

(K) $A \Rightarrow B \Rightarrow A$,

(S) $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$.

By "axiom schemes", we mean that the above formulas can be assumed for any given formulas A, B and C. In other words, this amounts to adding the rules

$$\frac{}{\Gamma \vdash A \Rightarrow B \Rightarrow A} \; (\mathrm{K}) \qquad \frac{}{\Gamma \vdash (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C} \; (\mathrm{S})$$

A sequent is *provable* when it is the conclusion of a proof built from the above rules, and a formula A is provable when the sequent $\vdash A$ is provable.

Example 2.7.1.1. For any formula A, the formula $A \Rightarrow A$ is provable:

$$\frac{\dfrac{\dfrac{}{\vdash (A \Rightarrow (B \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow B \Rightarrow A) \Rightarrow A \Rightarrow A} \; (\mathrm{S}) \quad \dfrac{}{\vdash A \Rightarrow (B \Rightarrow A) \Rightarrow A} \; (\mathrm{K})}{\vdash (A \Rightarrow B \Rightarrow A) \Rightarrow A \Rightarrow A} \; (\Rightarrow_\mathrm{E}) \quad \dfrac{}{\vdash A \Rightarrow B \Rightarrow A} \; (\mathrm{K})}{\vdash A \Rightarrow A} \; (\Rightarrow_\mathrm{E})$$

Note the complexity compared to NJ or LJ.

None of the rules modify the context Γ, so that people generally omit writing it. Also, traditionally, instead of using proof trees, a *proof* of A in the context Γ is formalized instead as a finite sequence of formulas $A_1, \ldots, A_n$, with $A_n = A$ such that either

– A_i belongs to Γ, or

– A_i is an instance of an axiom, or

– there are indices $j, k < i$ such that $A_k = A_j \Rightarrow A_i$, i.e. A_i can be deduced by

$$\frac{\Gamma \vdash A_j \Rightarrow A_i \qquad \Gamma \vdash A_j}{\Gamma \vdash A_i} \ (\Rightarrow_{\mathrm{E}})$$

This corresponds to describing the proof tree by some traversal of it.

Example 2.7.1.2. The proof example 2.7.1.1 is generally written as follows:

1. $(A \Rightarrow (B \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow B \Rightarrow A) \Rightarrow A \Rightarrow A$ by (S)

2. $A \Rightarrow (B \Rightarrow A) \Rightarrow A$ by (K)

3. $(A \Rightarrow B \Rightarrow A) \Rightarrow A \Rightarrow A$ by modus ponens on 1. and 2.

4. $A \Rightarrow B \Rightarrow A$ by (K)

5. $A \Rightarrow A$ by modus ponens on 3. and 4.

2.7.2 Other connectives. In the case where connectives other than implication are considered, appropriate axioms should be added:

conjunction:	$A \wedge B \Rightarrow A$	$A \Rightarrow B \Rightarrow A \wedge B$
	$A \wedge B \Rightarrow B$	
truth:		$A \Rightarrow \top$
disjunction:	$A \vee B \Rightarrow (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C$	$A \Rightarrow A \vee B$
		$B \Rightarrow A \vee B$
falsity:	$\bot \Rightarrow A$	
negation:	$\neg A \Rightarrow A \Rightarrow \bot$	$A \Rightarrow \bot \Rightarrow \neg A$

It can be observed that the axioms are in correspondence with elimination and introduction rules in natural deduction (respectively left and right column above). The classical variants of the system can be obtained by further adding one of the axioms from theorem 2.5.1.1.

2.7.3 Relationship with natural deduction. In order to show that proofs in Hilbert calculus correspond to proofs in natural deduction, we first need to study some of its properties. The usual structural rules are admissible in this system:

Proposition 2.7.3.1. The rules of exchange, contraction, truth strengthening and weakening are admissible in Hilbert calculus:

$$\frac{\Gamma, A, B, \Gamma' \vdash C}{\Gamma, B, A, \Gamma' \vdash C} \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \qquad \frac{\Gamma, \top \vdash A}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash C}{\Gamma, A \vdash C}$$

Proof. By induction on the proof of the premise. □

The introduction rule for implication is also admissible. This is sometimes called the *deduction theorem* and is due to Herbrand.

Proposition 2.7.3.2. The introduction rule for implication is admissible:

$$\frac{\Gamma, A, \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A \Rightarrow B} \ (\Rightarrow_{\mathrm{I}})$$

Proof. By induction on the proof of $\Gamma, A, \Gamma' \vdash B$.

– If it is of the form

$$\frac{}{\Gamma, A, \Gamma' \vdash A} \ (\mathrm{ax})$$

then we can show $\vdash A \Rightarrow A$ by example 2.7.1.1 and thus $\Gamma, \Gamma' \vdash A \Rightarrow A$ by weakening.

– If it is of the form

$$\frac{}{\Gamma, A, \Gamma' \vdash B} \ (\mathrm{ax})$$

with B different from A which belongs to Γ or Γ', then we can show $B \vdash A \Rightarrow B$ by

$$\frac{\dfrac{}{B \vdash B \Rightarrow A \Rightarrow B} \ (\mathrm{K}) \qquad \dfrac{}{B \vdash B} \ (\mathrm{ax})}{B \vdash A \Rightarrow B} \ (\Rightarrow_{\mathrm{E}})$$

and thus $\Gamma, \Gamma' \vdash A \Rightarrow B$ by weakening.

– If it is of the form

$$\frac{\Gamma, A, \Gamma' \vdash C \qquad \Gamma, A, \Gamma' \vdash C \Rightarrow B}{\Gamma, A, \Gamma' \vdash B} \ (\Rightarrow_{\mathrm{E}})$$

then, by induction hypothesis, we have proofs of $\Gamma, \Gamma' \vdash A \Rightarrow C$ and of $\Gamma, \Gamma' \vdash A \Rightarrow C \Rightarrow B$ and the derivation

$$\frac{\dfrac{\dfrac{}{\Gamma, \Gamma' \vdash (A \Rightarrow C \Rightarrow B) \Rightarrow (A \Rightarrow C) \Rightarrow A \Rightarrow B} \ (\mathrm{S}) \quad \dfrac{\vdots}{\Gamma, \Gamma' \vdash A \Rightarrow C \Rightarrow B}}{\Gamma, \Gamma' \vdash (A \Rightarrow C) \Rightarrow A \Rightarrow B} \ (\Rightarrow_{\mathrm{E}}) \quad \dfrac{\vdots}{\Gamma, \Gamma' \vdash A \Rightarrow C}}{\Gamma, \Gamma' \vdash A \Rightarrow B} \ (\Rightarrow_{\mathrm{E}})$$

allows us to conclude. □

We can thus show that provability in this system is the usual one.

Theorem 2.7.3.3. A sequent $\Gamma \vdash A$ is provable in Hilbert calculus if and only if it is provable in natural deduction.

Proof. For simplicity, we restrict to the case of the implicational fragment. In order to show that a proof in the Hilbert calculus induces a proof in NJ, we should show that the rules (ax) and $(\Rightarrow_{\mathrm{E}})$ are admissible in NJ (this is the case by definition) and that the axioms (S) and (K) can be derived in NJ, which is easy:

$$\frac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \Rightarrow C, A \Rightarrow B, A \vdash A \Rightarrow B \Rightarrow C} \ (\mathrm{ax}) \quad \dfrac{}{A \Rightarrow B \Rightarrow C, A \Rightarrow B, A \vdash A} \ (\mathrm{ax})}{A \Rightarrow B \Rightarrow C, A \Rightarrow B, A \vdash B \Rightarrow C} \ (\Rightarrow_{\mathrm{E}}) \quad \dfrac{\dfrac{}{A \Rightarrow B \Rightarrow C, A \Rightarrow B, A \vdash A \Rightarrow B} \ (\mathrm{ax}) \quad \dfrac{}{A \Rightarrow B \Rightarrow C, A \Rightarrow B, A \vdash A} \ (\mathrm{ax})}{A \Rightarrow B \Rightarrow C, A \Rightarrow B, A \vdash B} \ (\Rightarrow_{\mathrm{E}})}{\dfrac{\dfrac{A \Rightarrow B \Rightarrow C, A \Rightarrow B, A \vdash C}{A \Rightarrow B \Rightarrow C, A \Rightarrow B \vdash A \Rightarrow C}}{\dfrac{A \Rightarrow B \Rightarrow C \vdash (A \Rightarrow B) \Rightarrow A \Rightarrow C}{\vdash (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C}}} } \begin{array}{l} (\Rightarrow_{\mathrm{E}}) \\ (\Rightarrow_{\mathrm{I}}) \\ (\Rightarrow_{\mathrm{I}}) \\ (\Rightarrow_{\mathrm{I}}) \end{array}$$

(this is deliberately too small to read, you should prove this by yourself) and

$$\frac{\dfrac{\overline{A, B \vdash A}\ (\mathrm{ax})}{A \vdash B \Rightarrow A}\ (\Rightarrow_{\mathrm{I}})}{\vdash A \Rightarrow B \Rightarrow A}\ (\Rightarrow_{\mathrm{I}})$$

Conversely, in order to show that a proof in NJ induces one in Hilbert calculus, we should show that the rules (ax), ($\Rightarrow_{\mathrm{E}}$) and ($\Rightarrow_{\mathrm{I}}$) are admissible in Hilbert calculus: the first two are by definition, and the third one was proved in proposition 2.7.3.2. □

2.8 Kripke semantics

We have seen in section 2.5.6 that the usual boolean interpretation of formulas is correct and complete, meaning that a formula is classically provable if and only if it is valid in every boolean model. One can wonder if there is an analogous notion of model for proofs in intuitionistic logic – and this is indeed the case: Kripke models are correct and complete for intuitionistic logic. They were discovered in the 1960s by Kripke [Kri65] and Joyal for modal logic, and can be thought of as a semantics of possible worlds evolving through time: as time progresses more propositions may become true. The moral is thus that intuitionistic logic is a logic where the notion of truth is "local", unlike classical logic.

2.8.1 Kripke structures. A *Kripke structure* $(W, \leqslant, \rho)$ consists of a partially ordered set $(W, \leqslant)$ of *worlds* together with a *valuation* $\rho : W \times \mathcal{X} \to \mathbb{B}$ which indicates whether in a given world a given propositional variable is true or not. The valuation is always assumed to be *monotonous*, i.e. to satisfy $\rho(w, X) = 1$ implies $\rho(w', X) = 1$ for every worlds such that $w \leqslant w'$. We sometimes simply write W for a Kripke structure $(W, \leqslant, \rho)$.

Given a Kripke structure W and a world $w \in W$, we write $w \vDash_W A$ (or simply $w \vDash A$ when W is clear from the context) when a formula A is *satisfied* in w. This relation is defined by induction on A:

$$
\begin{aligned}
&w \vDash X && \text{iff } \rho(w, X) \\
&w \vDash \top && \text{holds} \\
&w \vDash \bot && \text{does not hold} \\
&w \vDash A \wedge B && \text{iff } w \vDash A \text{ and } w \vDash B \\
&w \vDash A \vee B && \text{iff } w \vDash A \text{ or } w \vDash B \\
&w \vDash A \Rightarrow B && \text{iff, for every } w' \geqslant w,\ w' \vDash A \text{ implies } w' \vDash B \\
&w \vDash \neg A && \text{iff, for every } w' \geqslant w,\ w' \vDash A \text{ does not hold}
\end{aligned}
$$

A Kripke structure is often pictured as a graph whose vertices correspond to worlds, an edge from w to w' indicating that $w \leqslant w'$, with the variables X such that $\rho(w, X) = 1$ being written next to the node w, see examples 2.8.1.4 and 2.8.1.5.

We can think of a Kripke structure as describing the evolution of a world through time: given two worlds such that $w \leqslant w'$, we think of w' as being a possible future for w. Since the order is not necessarily total, a given world might have different possible futures. In each world, the valuation indicates

which formulas we know are true, and the monotonicity condition ensures that our knowledge can only grow: if we know that a formula is true then we will still know it in the future.

Lemma 2.8.1.1. Satisfaction is monotonic: given a formula A, a Kripke structure W and a world w, if $w \vDash A$ then $w' \vDash A$ for every world $w' \geqslant w$.

Proof. By induction on the formula A. □

Given a context $\Gamma = A_1, \ldots, A_n$, a formula A, and a Kripke structure W, we write $\Gamma \vDash_W A$ when, for every world $w \in W$ in which all the formulas A_i are satisfied, the formula A is also satisfied. We write $\Gamma \vDash A$ when $\Gamma \vDash_W A$ holds for every structure W: in this case, we say that A is *valid* in the context Γ.

Remark 2.8.1.2. It should be observed that the notion of Kripke structure generalizes the notion of boolean model recalled in section 2.5.6. Namely, a boolean valuation $\rho : \mathcal{X} \to \mathbb{B}$ can be seen as a Kripke structure W with a single world w, the valuation being given by ρ. The notion of validity for Kripke structures defined above then coincides with the one for boolean models.

A following theorem ensures that Kripke semantics are sound: a provable formula is valid.

Theorem 2.8.1.3 (Soundness). If a sequent $\Gamma \vdash A$ is derivable in intuitionistic logic then $\Gamma \vDash A$.

Proof. By induction on the proof of $\Gamma \vdash A$. □

The contrapositive of this theorem says that if we can find a Kripke structure in which there is a world where a formula A is not satisfied, then A is not intuitionistically provable. This thus provides an alternative to methods based on cut-elimination (see section 2.3.5) in order to establish the non-provability of formulas.

Example 2.8.1.4. Consider the formula expressing double negation elimination $\neg\neg X \Rightarrow X$ and the Kripke structure with $W = \{w_0, w_1\}$, with $w_0 \leqslant w_1$, and $\rho(w_0, X) = 0$ and $\rho(w_1, X) = 1$, which can be pictured as

$$\underset{w_0}{\cdot} \longrightarrow \overset{X}{\underset{w_1}{\cdot}}$$

We have $w_0 \nvDash \neg X$ (because there is the future world w_1 in which X holds) and $w_1 \nvDash \neg X$, and thus $w_0 \vDash \neg\neg X$ (in fact, it can be shown that $w \vDash \neg\neg X$ in an arbitrary structure iff for every world $w' \geqslant w$ there exists a world $w'' \geqslant w'$ such that $w'' \vDash X$). Moreover, we have $w_0 \nvDash X$, and thus $w_0 \nvDash \neg\neg X \Rightarrow X$. This shows that $\neg\neg X \Rightarrow X$ is not intuitionistically provable. In the same Kripke structure, we have $w_0 \nvDash \neg X \vee X$ and thus the excluded middle $\neg X \vee X$ is not intuitionistically provable either.

Given an arbitrary formula A, by lemma 2.8.1.1, in this structure, this formula is either satisfied both in w_0 and w_1, or only in w_1 or in no world:

$$\overset{A}{\underset{w_0}{\cdot}} \longrightarrow \overset{A}{\underset{w_1}{\cdot}} \qquad \underset{w_0}{\cdot} \longrightarrow \overset{A}{\underset{w_1}{\cdot}} \qquad \underset{w_0}{\cdot} \longrightarrow \underset{w_1}{\cdot}$$

In the two first cases, $\neg\neg A$ is satisfied and in the last one $\neg A$ is satisfied. Therefore, the weak excluded middle $\neg\neg A \vee \neg A$ is always satisfied: this shows

that the weak excluded middle does not imply the excluded middle. By using a similar reasoning, it can be shown that the linearity axiom $(A \Rightarrow B) \vee (B \Rightarrow A)$ does not imply the excluded middle. Both thus give rise to intermediate logics, see section 2.5.10.

Example 2.8.1.5. The Kripke structure

$$\overset{X}{\bullet} \longleftarrow \cdot \longrightarrow \overset{Y}{\bullet}$$

shows that the linearity formula $(X \Rightarrow Y) \vee (Y \Rightarrow X)$ is not intuitionistically provable (whereas classically it is).

2.8.2 Completeness. We now consider the converse of the theorem 2.8.1.3. We will show that if a formula is valid then it is provable intuitionistically. Or equivalently, that if a formula is not provable, then we can always exhibit a Kripke structure in which it does not hold.

Given a possibly infinite set Φ of formulas, we write $\Phi \vdash A$ whenever there exists a finite subset $\Gamma \subseteq \Phi$ such that $\Gamma \vdash A$ is intuitionistically provable. Such a set is *consistent* if $\Phi \nvdash \bot$. By lemma 2.3.4.1, we have:

Lemma 2.8.2.1. A set Φ of formulas is consistent if and only if there is a formula A such that $\Phi \nvdash A$.

A set Φ of formulas is *disjunctive* if $\Phi \vdash A \vee B$ implies $\Phi \vdash A$ or $\Phi \vdash B$.

Lemma 2.8.2.2. Given a set Φ of formulas and a formula A such that $\Phi \nvdash A$, there exists a disjunctive set $\overline{\Phi}$ such that $\Phi \subseteq \overline{\Phi}$ and $\overline{\Phi} \nvdash A$.

Proof. Suppose fixed an enumeration of all the formulas of the form $B \vee C$ occurring in Φ. We construct by induction a sequence Φ_n of sets of formulas which are such that $\Phi_n \nvdash A$. We set $\Phi_0 = \Phi$. Suppose Φ_n constructed and consider the n-th formula $B \vee C$ in Φ:

  – if $\Phi_n, B \nvdash A$, we define $\Phi_{n+1} = \Phi_n \cup \{B\}$,

  – if $\Phi_n, B \vdash A$, we define $\Phi_{n+1} = \Phi_n \cup \{C\}$.

In the first case, it is obvious that $\Phi_{n+1} \nvdash A$. In the second one, we have $\Phi_n \vdash B \vee C$ and $\Phi_n, B \vdash A$, if we also had $\Phi_n, C \vdash A$ then, by $(\vee_{\mathrm{E}})$, we would also have $\Phi_n \vdash A$, which is excluded by induction hypothesis. Finally, we take $\overline{\Phi} = \bigcup_{n \in \mathbb{N}} \Phi_n$. □

A set Φ of formulas is *saturated* if, for every formula A, $\Phi \vdash A$ implies $A \in \Phi$.

Lemma 2.8.2.3. Given a set Φ of formulas, the set $\overline{\Phi} = \{A \mid \Phi \vdash A\}$ is saturated.

A set is *complete* if it is consistent, disjunctive and saturated. Combining the above lemmas we obtain,

Lemma 2.8.2.4. Given a set Φ of formulas and a formula A such that $\Phi \nvdash A$, there exists a complete set $\overline{\Phi}$ such that $\Phi \subseteq \overline{\Phi}$ and $A \notin \overline{\Phi}$.

Proof. By lemma 2.8.2.2, there exists a disjunctive set of formulas $\overline{\Phi}$ such that $\Phi \subseteq \overline{\Phi}$ and $\overline{\Phi} \nvdash A$. This set of consistent by lemma 2.8.2.1. Moreover, by lemma 2.8.2.3, we can suppose that this set is saturated (the construction is easily checked to preserve consistency and disjunctiveness) and such that $A \notin \overline{\Phi}$. □

The *universal Kripke structure W* is defined by

$$W = \{w_\Phi \mid \Phi \text{ is complete}\}$$

with $w_\Phi \leqslant w_{\Phi'}$ whenever $\Phi \subseteq \Phi'$, and $\rho(w_\Phi, X) = 1$ iff $X \in \Phi$.

Lemma 2.8.2.5. Let Φ be a complete set and A a formula. Then $w_\Phi \vDash A$ iff $A \in \Phi$.

Proof. By induction on the formula A.

– If $A = X$ is a propositional variable, we have $w_\Phi \vDash X$ iff $\rho(w_\Phi, X) = 1$ iff $X \in \Phi$.

– If $A = \top$, we always have $w_\Phi \vDash \top$ and we always have $\top \in \Phi$ because $\Phi \vdash \top$ by $(\top_I)$ and Φ is saturated.

– If $A = \bot$, we never have $w_\Phi \vDash \bot$ and we never have $\bot \in \Phi$ because Φ is consistent.

– If $A = B \wedge C$.
Suppose that $w_\Phi \vDash B \wedge C$. Then $w_\Phi \vDash B$ and $w_\Phi \vDash C$, and therefore $\Phi \vdash B$ and $\Phi \vdash C$ by induction hypothesis. We deduce $\Phi \vdash B \wedge C$ by $(\wedge_I)$ and weakening and thus $B \wedge C \in \Phi$ by saturation.
Conversely, suppose $B \wedge C \in \Phi$ and thus $\Phi \vdash B \wedge C$. This entails $\Phi \vdash B$ and $\Phi \vdash C$ by $(\wedge_E^l)$ and $(\wedge_E^r)$, and thus $B \in \Phi$ and $C \in \Phi$ by saturation. By induction hypothesis, we have $w_\Phi \vDash B$ and $w_\Phi \vDash C$, and thus $w_\Phi \vDash B \wedge C$.

– If $A = B \vee C$.
Suppose that $w_\Phi \vDash B \vee C$. Then $w_\Phi \vDash B$ or $w_\Phi \vDash C$, and therefore $\Phi \vdash B$ or $\Phi \vdash C$ by induction hypothesis. We deduce $\Phi \vdash B \vee C$ by $(\vee_I^l)$ or $(\vee_I^r)$ and thus $B \vee C \in \Phi$ by saturation.
Conversely, suppose $B \vee C \in \Phi$. Since Φ is disjunctive, we have $B \in \Phi$ or $C \in \Phi$. By induction hypothesis, we have $w_\Phi \vDash B$ or $w_\Phi \vDash C$, and thus $w_\Phi \vDash B \vee C$.

– If $A = B \Rightarrow C$.
Suppose that $w_\Phi \vDash B \Rightarrow C$. Our goal is to show $B \Rightarrow C \in \Phi$. By $(\Rightarrow_I)$ and saturation, it is enough to show $\Phi, B \vdash C$. Suppose that it is not the case. By lemma 2.8.2.4, we can construct a complete set Φ' with $\Phi \subseteq \Phi'$, $B \in \Phi'$ and $C \notin \Phi'$. Since $B \in \Phi'$, by induction hypothesis we have $w_{\Phi'} \vDash B$. Therefore, since $w_\Phi \vDash B \Rightarrow C$ and $w_\Phi \leqslant w_{\Phi'}$, we have $w_{\Phi'} \vDash C$, a contradiction.
Conversely, suppose $B \Rightarrow C \in \Phi$. Given Φ' such that $w_\Phi \leqslant w_{\Phi'}$, i.e. $\Phi \subseteq \Phi'$, if $w_{\Phi'} \vDash B$ we have to show $w_{\Phi'} \vDash C$. By induction hypothesis, we have $B \in \Phi'$. Moreover, we have $\Phi' \vdash B \Rightarrow C$ (because $\Phi \vdash B \Rightarrow C$ and $\Phi \subseteq \Phi'$) and therefore $\Phi' \vdash C$ by $(\Rightarrow_E)$. By saturation we have $C \in \Phi'$ and thus $\Phi' \vDash C$, by induction hypothesis. □

Theorem 2.8.2.6 (Completeness). If $\Gamma \vDash A$ then $\Gamma \vdash A$.

Proof. Suppose $\Gamma \vDash A$ and $\Gamma \nvdash A$. By lemma 2.8.2.4, there exists a complete set of formulas Φ such that $\Gamma \subseteq \Phi$ and $\Phi \nvdash A$. All the formulas of Γ are valid in w_Φ and thus $w_\Phi \vDash A$, because we have $\Gamma \vDash A$. By lemma 2.8.2.5, we therefore have $A \in \Phi$, a contradiction. □

Remark 2.8.2.7. It can be shown that we can restrict to Kripke models which are tree-shaped and finite without losing completeness. With further restrictions, various completeness results have been obtained. As an extreme example, if we restrict to models with only one world, then we obtain boolean models (remark 2.8.1.2) which are complete for classical logic (theorem 2.5.6.5). For a more unexpected example, Kripke models which are total orders are complete for intuitionistic logic extended with the linearity axiom (section 2.5.10), thus its name.

Pure λ-calculus

We now introduce the λ-calculus, which is the functional core of a programming language: this is what you obtain when you remove everything from a functional programming language except for the variables, functions and application. In this language everything is thus a function. In the OCaml syntax, a typical λ-term would thus be

```
fun f x -> f (f x) (fun y -> y)
```

Since λ-calculus was actually invented before computers existed, the traditional notation is somewhat different from the above, and we write $\lambda x.t$ instead of `fun x -> t` so that the above term would rather be written

$$\lambda f x.f(fx)(\lambda y.y)$$

Bound variables. In a function, the name of the variable is not important, it could be replaced by any other name without changing the meaning of the function: we should consider $\lambda x.x$ and $\lambda y.y$ as the same. In a term of the form $\lambda x.t$, we say that the abstraction λ *binds* the variable in the term t: the name of the variable x in t is not really relevant, what matters is that this is the variable which was declared by this λ. In mathematics, we are somewhat used to this in other situations than functions. For instance, in the first definition below, t is bound by the limit operator, in the second t is bound by the $\mathrm{d}t$ operator coming with the integral, and in the last one the summation sign is binding i:

$$f(x) = \lim_{t \to \infty} \frac{x}{t} \qquad f(x) = \int_0^1 tx \, \mathrm{d}t \qquad f(x) = \sum_{i=0}^n ix$$

This means that we can replace the name of the bound variable by any other (as above) without changing the meaning of the expression. For instance, the first one is equivalent to

$$\lim_{z \to \infty} \frac{x}{z}$$

This process of changing the name of the variable is called α-*conversion* and is more subtle than it seems at first: there are actually some restrictions on the names of the variables we can use. For instance, in the above example, we cannot rename t to x since the following reasoning is clearly not valid:

$$0 = \lim_{t \to \infty} \frac{x}{t} = \lim_{x \to \infty} \frac{x}{x} = \lim_{x \to \infty} 1 = 1$$

The problem here is that we tried to change the name of t to a variable name which was already used somewhere else. These issues are generally glossed over in mathematics, but in computer science we cannot simply do that: we have to understand in details these α-conversion mechanisms when implementing functional programming languages, otherwise we will evaluate programs incorrectly. Believe it or not this simple matter is a major source of bugs and headaches.

Evaluation. Another aspect we have to make precise is the notion of evaluation or *reduction* in a functional programming language. In mathematics, if f is the doubling function $f(x) = x + x$, then $f(3)$ is $3 + 3$, i.e. 6: with our λ notation, we have $(\lambda x.x + x)3 = 6$. In computer science, we want to see the way the program is executed and we will consider that $(\lambda x.x + x)3$ reduces to $3 + 3$, which will itself reduce to 6, which is the result of our program. The general definition of the reduction in the language is given by the β-*reduction* rule, which is

$$(\lambda x.t)u \longrightarrow_\beta t[u/x]$$

It means that the function which to x associates some expression t, when applied to an argument u reduces to t where all the occurrences of x have been replaced by u. The properties of this reduction relation is one of our main objects of interest here.

In this chapter. We introduce the λ-calculus in section 3.1 and the β-reduction in section 3.2. We then study the computational power of the resulting calculus in section 3.3 and show that reduction is confluent in section 3.4. We discuss the various ways in which reduction can be implemented in section 3.5, and the ways to handle α-conversion in section 3.6.

References. Should you need a more detailed presentation of λ-calculus, its properties and applications, good introductions include [Bar84, SU06, Sel08].

3.1 λ-terms

3.1.1 Definition. Suppose fixed an infinite countable set $\mathcal{X} = \{x, y, z, \ldots\}$ whose elements are called *variables*. The set Λ of λ-*terms* is generated by the following grammar:

$$t, u ::= x \mid t\,u \mid \lambda x.t$$

This means that a λ-term is either

- a *variable x*,

- an *application $t\,u$*, which is a pair of terms t and u, thought of as applying the function t to an argument u,

- an *abstraction $\lambda x.t$*, which is a pair consisting of a variable x and a term t, thought of as the function which to x associates t.

For instance, we have the following λ-terms

$$\lambda x.x \qquad (\lambda x.(xx))(\lambda y.(yx)) \qquad \lambda x.(\lambda y.(x(\lambda z.y)))$$

By convention,

- application is associative to the left, i.e.

$$tuv = (tu)v$$

and not $t(uv)$,

– application binds more tightly than abstraction, i.e.

$$\lambda x.xy = \lambda x.(xy)$$

and not $(\lambda x.x)y$ (in other words, abstraction extends as far as possible on the right),

– we sometimes group abstractions, i.e.

$$\lambda xyz.xz(yz) \qquad \text{is read as} \qquad \lambda x.\lambda y.\lambda z.xz(yz).$$

3.1.2 Bound and free variables. In a term of the form $\lambda x.t$, the variable x is said to be *bound* in the term t: in a sense the abstraction "declares" the variable in the term t, and all occurrences of x in t will make reference to the variable declared here (unless it is bound again). Thus, in the term $(\lambda x.xy)x$, the first occurrence of x refers to the variable declared by the abstraction whereas the second does not. Intuitively, this term is the same as $(\lambda z.zy)x$, but not as $(\lambda z.zy)z$; this will be made formal below through the notion of α-equivalence, but we should keep in mind that there is always the possibility of renaming bound variables.

A *free variable* in a term is a variable which is not bound in a subterm. We define the set $\mathrm{FV}(t)$ of a term t, by induction on t, by

$$
\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(t\,u) &= \mathrm{FV}(t) \cup \mathrm{FV}(u) \\
\mathrm{FV}(\lambda x.t) &= \mathrm{FV}(t) \setminus \{x\}
\end{aligned}
$$

A term t is *closed* when it has no free variable: $\mathrm{FV}(t) = \emptyset$.

Example 3.1.2.1. The set of free variables of the term $(\lambda x.xy)z$ is $\{y, z\}$. This term is thus not closed. The term $\lambda xy.x$ is closed.

A variable x is *fresh* with respect to a term t when it does not occur as a free variable in t, i.e. $x \in \mathcal{X} \setminus \mathrm{FV}(t)$. Note that the set of variables of a term t is finite and the set of variable is infinite so that we can always find a fresh variable with respect to any term.

3.1.3 Renaming and α-equivalence. In order to define α-equivalence, we first define the operation of renaming a variable x to y in a term t, and write

$$t\{y/x\}$$

for the resulting term. There is one subtlety though, we only want to rename free occurrences of x, since the other ones refer to the abstraction to which they are bound. Formally, the *renaming* $t\{y/x\}$ is defined by

$$
\begin{aligned}
x\{y/x\} &= y & \\
z\{y/x\} &= z & \text{if } z \neq x \\
(t\,u)\{y/x\} &= (t\{y/x\})\,(u\{y/x\}) & \\
(\lambda x.t)\{y/x\} &= \lambda x.t & \\
(\lambda z.t)\{y/x\} &= \lambda z.(t\{y/x\}) & \text{if } z \neq x \text{ and } z \neq y \\
(\lambda y.t)\{y/x\} &= \lambda z.(t\{z/y\}\{y/x\}) & \text{for some } z \text{ with } z \notin \mathrm{FV}(t) \cup \{x, y\}
\end{aligned}
$$

The three last lines handle the possible cases when renaming a variable in an abstraction: either we are trying to rename the bound variable, or the bound variable and variables involved in the renaming are distinct, or we are trying to rename a variable into the bound variable.

The α-*equivalence* $=\!\!=_\alpha$ (or α-*conversion*) is the smallest congruence (see below) on terms which identifies terms differing only by renaming bound variables, i.e.

$$\lambda x.t =\!\!=_\alpha \lambda y.(t\{y/x\})$$

whenever y is not free in t. For instance, we have

$$\lambda x.x(\lambda x.xy) =\!\!=_\alpha \lambda z.z(\lambda x.xy) =\!\!\!\!/\!=_\alpha \lambda y.y(\lambda x.xy)$$

Formally, the fact that α-equivalence is a congruence means that it is the smallest relation such that whenever all the relations above the bar hold, the relation below the bar also holds:

$$\frac{y \notin \mathrm{FV}(t)}{\lambda x.t =\!\!=_\alpha \lambda y.(t\{y/x\})}$$

$$\frac{t =\!\!=_\alpha t' \qquad u =\!\!=_\alpha u'}{t\,u =\!\!=_\alpha t'\,u'} \qquad\qquad \frac{t =\!\!=_\alpha t'}{\lambda x.t =\!\!=_\alpha \lambda x.t'}$$

$$\frac{}{t =\!\!=_\alpha t} \qquad \frac{t =\!\!=_\alpha t'}{t' =\!\!=_\alpha t} \qquad \frac{t =\!\!=_\alpha t' \qquad t' =\!\!=_\alpha t''}{t =\!\!=_\alpha t''}$$

The equation on the first line is the one we have already seen above, those on the second line ensure that α-equivalence is compatible with application and abstraction, and those on the third line impose that α-equivalence is an equivalence relation (i.e. reflexive, symmetric and transitive).

3.1.4 Substitution. Given λ-terms t and u and a variable x, we can define a new term

$$t[u/x]$$

which is the λ-term obtained from t by replacing free occurrences of x by u. Again we have to properly take care of issues related to the fact that some variables are bound:

– we only want to replace free occurrences of the variable x in t, since the bound ones refer to the corresponding abstractions in t and might be renamed, i.e.

$$(x(\lambda xy.x))[u/x] = u(\lambda xy.x) \qquad \text{but not } u(\lambda xy.u),$$

– we do not want free variables in u to become accidentally bound by some abstraction in t, i.e.

$$(\lambda x.xy)[x/y] = (\lambda z.zy)[x/y] = \lambda z.zx \qquad \text{but not } \lambda x.xx.$$

Formally, the *substitution* $t[u/x]$ is defined by induction on t by

$$x[u/x] = u$$
$$y[u/x] = y \qquad \text{if } y \neq x$$
$$(t_1\, t_2)[u/x] = (t_1[u/x])\,(t_2[u/x])$$
$$(\lambda x.t)[u/x] = \lambda x.t$$
$$(\lambda y.t)[u/x] = \lambda y.(t[u/x]) \qquad \text{if } y \neq x \text{ and } y \notin \text{FV}(u)$$
$$(\lambda y.t)[u/x] = \lambda y'.(t\{y'/y\}[u/x]) \qquad \text{if } y \neq x,\ y \in \text{FV}(u)$$
$$\text{and } y' \notin \text{FV}(t) \cup \text{FV}(u) \cup \{x\}.$$

Because of the last line, the result of the substitution is not well-defined, because it depends on an arbitrary choice of a fresh variable y', but one can show that this is a well-defined operation on λ-terms up to α-equivalence. For this reason, as soon as we want to perform substitutions, it only makes sense to consider the set of λ-terms quotiented by the α-equivalence relation: we will implicitly do so in the following, and implicitly ensure that all the constructions we perform are compatible with α-equivalence. The only time where we should take α-conversion seriously is when dealing with implementation matters, see section 3.6.2 for instance. Adopting this convention, the three last cases can be replaced by

$$(\lambda y.t)[u/x] = \lambda y.(t[u/x])$$

where we suppose that $y \notin \text{FV}(t) \cup \{x\}$, which we can always do up to α-conversion.

3.2 β-reduction

Consider a term of the form

$$(\lambda x.t)\, u \tag{3.1}$$

It intuitively consists of a function expecting an argument x and returning a result $t(x)$, which is given an argument u. We expect therefore the computation to reach the term $t[u/x]$ consisting of the term t where all the free occurrences of x have been replaced by u. This is what the notion of β-reduction does and we write

$$(\lambda x.t)\, u \longrightarrow_\beta t[u/x] \tag{3.2}$$

to indicate that the term on the left reduces to the term on the right. Actually, we want to be able to also perform this kind of reduction within a term: we call a *β-redex* in a term t, a subterm of the form (3.1) and the β-reduction consists in preforming the replacement (3.2) in that term.

3.2.1 Definition. Formally, the *β-reduction* is defined as the smallest binary relation $\longrightarrow_\beta$ on terms such that

$$\frac{}{(\lambda x.t)u \longrightarrow_\beta t[u/x]}\ (\beta_{\mathrm{s}}) \qquad\qquad \frac{t \longrightarrow_\beta t'}{\lambda x.t \longrightarrow_\beta \lambda x.t'}\ (\beta_\lambda)$$

$$\frac{t \longrightarrow_\beta t'}{tu \longrightarrow_\beta t'u}\ (\beta_{\mathrm{l}}) \qquad\qquad \frac{u \longrightarrow_\beta u'}{tu \longrightarrow_\beta tu'}\ (\beta_{\mathrm{r}})$$

A "proof tree" showing that $t \longrightarrow_\beta u$ is called a *derivation* of it. For instance, a derivation of $\lambda x.(\lambda y.y)xz \longrightarrow_\beta \lambda x.xz$ is

$$\dfrac{\dfrac{\overline{(\lambda y.y)x \longrightarrow_\beta x}\ \ (\beta_{\mathsf{s}})}{(\lambda y.y)xz \longrightarrow_\beta xz}\ \ (\beta_{\mathsf{l}})}{\lambda x.(\lambda y.y)xz \longrightarrow_\beta \lambda x.xz}\ \ (\beta_\lambda)$$

Such derivations are often useful to reason about β-reduction steps, by induction on the derivation tree.

3.2.2 An example. For instance, we have the following sequence of β-reductions, were each time we have underlined the β-redex:

$$\begin{aligned}(\lambda x.y)(\underline{(\lambda z.zz)(\lambda t.t)}) &\longrightarrow_\beta (\lambda x.y)(\underline{(\lambda t.t)(\lambda t.t)}) \\ &\longrightarrow_\beta \underline{(\lambda x.y)(\lambda t.t)} \\ &\longrightarrow_\beta y\end{aligned}$$

3.2.3 Reduction and redexes. Let us now make some basic observations about how reductions interact with redexes. Reduction can create β-redexes:

$$(\lambda x.xx)(\lambda y.y) \longrightarrow_\beta (\lambda y.y)(\lambda y.y)$$

In the initial term there was only one redex, and after reducing it a new redex has appeared. Reductions can duplicate β-redexes:

$$(\lambda x.xx)((\lambda y.y)(\lambda z.z)) \longrightarrow_\beta ((\lambda y.y)(\lambda z.z))((\lambda y.y)(\lambda z.z))$$

The β-redex $(\lambda y.y)(\lambda z.z)$ occurs once in the initial term and twice in the reduced one. Reduction can also erase β-redexes:

$$(\lambda x.y)((\lambda y.y)(\lambda z.z)) \longrightarrow_\beta y$$

There were two redexes in the initial term, but there is none left after reducing one of them.

3.2.4 Confluence. The reduction is not deterministic since some terms can reduce in multiple ways:

$$\lambda y.y \ {}_\beta\!\longleftarrow (\lambda xy.y)((\lambda x.x)(\lambda x.x)) \longrightarrow_\beta (\lambda xy.y)(\lambda x.x)$$

We thus have to be careful when studying properties of reduction: in particular, we always have to specify whether those properties hold for some reduction or every reduction. It can be noted that, although the two above reductions differ, they end up with the same term. For instance, the term on the right above reduces to $\lambda y.y$, which is the term on the left. This property is called "confluence": eventually, the order in which we chose to perform β-reductions does not matter. This will be detailed and proved in section 3.4.

3.2.5 β-reduction paths. A *reduction path*

$$t = t_0 \longrightarrow_\beta t_1 \longrightarrow_\beta t_2 \longrightarrow_\beta \ldots \longrightarrow_\beta t_{n-1} \longrightarrow_\beta t_n = u$$

from t to u is a finite sequence of terms $t_0, t_1, \ldots, t_n$ such that t_i β-reduces to t_{i+1} for every index i, with $t_0 = t$ and $t_n = u$. The natural number n is called the *length* of the reduction path. We write

$$t \overset{*}{\longrightarrow}_\beta u$$

when there exists a reduction path from t to u as above, and say that t reduces in multiple steps to u. The relation $\overset{*}{\longrightarrow}_\beta$ on terms is the reflexive and transitive closure of the relation $\longrightarrow_\beta$.

3.2.6 Normalization. Some terms cannot reduce, they are called *normal forms*:

$$x \qquad\qquad x(\lambda y.\lambda z.y) \qquad\qquad \ldots$$

Those can be though of as "values" or "results" for computations in λ-calculus. Those terms are easily characterized:

Proposition 3.2.6.1. The λ-terms in normal form can be characterized inductively as the terms of the form

$$\lambda x.t \qquad\quad \text{or} \qquad\quad x\, t_1 \ldots t_n$$

where the t_i and t are normal forms.

Proof. We reason by induction on the size of λ-terms (the size being the number of abstractions and applications). Suppose given a λ-term in normal form: by definition it can be of the following forms.

- – x: it is a normal form and it is a term generated of the expected form.

- – $\lambda x.t$: by the rule (β_λ), this term is a normal form if and only if t is, i.e. by induction, if and only if t is itself of the expected form.

- – $t\, u$: by the rules (β_l) and (β_r), if it is a normal form then both t and u are in normal form. By induction, t must be of the form $\lambda x.t'$ or $x\, t_1 \ldots t_n$ with t' and t_i of the expected form. The first case is impossible: otherwise, $t\, u = (\lambda x.t')u$ would reduce by (β_s). Therefore, $t\, u$ is of the form $x\, t_1 \ldots t_n\, u$ with t_i and u in normal form. Conversely, any term of this form is a normal form. $\qquad\square$

Having identified normal forms as the notion of "result" in the λ-calculus, it is natural to study whether every term will eventually give rise to a result; we will see that this is not the case. A term t is *weakly normalizing* when it can reduce to a normal form, i.e. there exists a normal form u such that $t \overset{*}{\longrightarrow}_\beta u$. It is *strongly normalizing* when every sequence of reductions will eventually reduce to a normal form. In other words, there is no infinite sequence of reductions starting from t:

$$t = t_0 \longrightarrow_\beta t_1 \longrightarrow_\beta t_2 \longrightarrow_\beta \ldots$$

Not every term is strongly normalizing. For instance, the term

$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

reduces to itself and thus infinitely:

$$(\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta (\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta (\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta \dots$$

As a variant, the following term keeps growing during the reduction:

$$(\lambda x.xx)(\lambda y.yyy) \longrightarrow_\beta (\lambda y.yyy)(\lambda y.yyy)$$
$$\longrightarrow_\beta (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \longrightarrow_\beta \dots$$

Clearly, a strongly normalizing term is weakly normalizing, but the converse does not hold. For instance, the term

$$(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$$

can reduce to y, which is a normal form, and is thus weakly normalizing. It can also reduce to itself and is thus not strongly normalizing.

3.2.7 β-equivalence. We write $==_\beta$ for the β-*equivalence*, which is the smallest equivalence relation containing $\longrightarrow_\beta$. It is not difficult to show that this relation can be characterized as the symmetric closure of the relation $\overset{*}{\longrightarrow}_\beta$: we have

$$t ==_\beta u$$

whenever there exists terms $t_0, \dots, t_{2n}$ such that

$$t = t_0 \underset{\beta}{\overset{*}{\longleftarrow}} t_1 \overset{*}{\longrightarrow}_\beta t_2 \underset{\beta}{\overset{*}{\longleftarrow}} t_3 \overset{*}{\longrightarrow}_\beta \dots {}_\beta \overset{*}{\longleftarrow} t_{2n-1} \overset{*}{\longrightarrow}_\beta t_{2n} = u$$

The notion of β-equivalence is very natural on λ-terms: it identifies two terms whenever they give rise to the same result. Two β-equivalent terms are sometimes also said to be β-*convertible*.

3.2.8 η-equivalence. In OCaml, the functions `sin` and `fun x -> sin x` are clearly "the same": one can be used in place of another without changing anything, both will compute the sine of their input. However, they are not identical: their syntax differ. In λ-calculus, the η-*equivalence* relation relates two such terms: it identifies a term t (which is a function, since everything is a function in λ-calculus) with the function which to x associates $t\,x$. Formally, the η-equivalence relation $==_\eta$ is the smallest congruence such that

$$t ==_\eta \lambda x.tx$$

for every term t.

By analogy with β-reduction, it will sometimes be useful to consider the η-*reduction* relation which is the smallest congruence such that

$$\lambda x.tx \longrightarrow_\eta t$$

for every term t. The opposite relation

$$t \longrightarrow_\eta \lambda x.tx$$

is also useful and called *η-expansion*. We have that η-equivalence is the reflexive, symmetric and transitive closure of this relation.

Finally, we write $==_{\beta\eta}$ for the *βη-equivalence* relation, which is smallest equivalence relation containing both $==_\beta$ and $==_\eta$. In this book, we will mostly focus on β-equivalence, although most proofs generalize to $\beta\eta$-equivalence.

3.3 Computing in the λ-calculus

The λ-calculus contains only functions. Even though we have removed most of what is usually found in a programming language, we will see that it is far from trivial as a programming language. In order to do so, we will gradually show that usual programming constructions can be encoded in λ-calculus.

3.3.1 Identity. A first interesting term is the *identity* λ-term

$$\mathsf{I} = \lambda x.x$$

It has the property that, for any term t, we have

$$\mathsf{I}\,t \longrightarrow_\beta t$$

3.3.2 Booleans. The booleans true and false can respectively be encoded as

$$\mathsf{T} = \lambda xy.x \qquad\qquad \mathsf{F} = \lambda xy.y$$

With this encoding, the usual if-then-else conditional construction can be encoded as

$$\mathsf{if} = \lambda bxy.bxy$$

Namely, we have

$$\mathsf{if}\,\mathsf{T}\,t\,u \stackrel{*}{\longrightarrow}_\beta t \qquad\qquad \mathsf{if}\,\mathsf{F}\,t\,u \stackrel{*}{\longrightarrow}_\beta u$$

For instance, the first reduction is

$$\begin{aligned}
\mathsf{if}\,\mathsf{T}\,t\,u = (\lambda bxy.bxy)(\lambda xy.x)tu &\longrightarrow_\beta (\lambda xy.(\lambda xy.x)xy)tu \\
&\longrightarrow_\beta (\lambda y.(\lambda xy.x)ty)u \\
&\longrightarrow_\beta (\lambda xy.x)tu \\
&\longrightarrow_\beta (\lambda y.t)u \\
&\longrightarrow_\beta t
\end{aligned}$$

and the second one is similar.

From there, the usual boolean operations of conjunction, disjunction and negation are easily defined by

$$\mathsf{and} = \lambda xy.x\,y\,\mathsf{F} \qquad \mathsf{or} = \lambda xy.x\,\mathsf{T}\,y \qquad \mathsf{not} = \lambda x.x\,\mathsf{F}\,\mathsf{T}$$

For instance, one can check that we have

$$\mathsf{and}\,\mathsf{T}\,\mathsf{T} \longrightarrow_\beta \mathsf{T} \quad \mathsf{and}\,\mathsf{T}\,\mathsf{F} \longrightarrow_\beta \mathsf{F} \quad \mathsf{and}\,\mathsf{F}\,\mathsf{T} \longrightarrow_\beta \mathsf{F} \quad \mathsf{and}\,\mathsf{F}\,\mathsf{F} \longrightarrow_\beta \mathsf{F}$$

Above, we have defined conjunction (and other operations) from conditionals, which is quite classical. In OCaml, we would have written

```
let and x y = if x then y else false
```

which translates in λ-calculus as

$$\text{and} \xleftarrow{*}_\eta \lambda xy.\text{and}\, x\, y = \lambda xy.\text{if}\, x\, y\, \mathsf{F} \xrightarrow{*}_\beta \lambda xy.\, x\, y\, \mathsf{F}$$

and suggests the definition we made. There are of course other possible implementations, e.g.

$$\text{and} = \lambda xy.xyx$$

In the above implementations, we only guarantee that the expected reductions will happen when the arguments are booleans, but nothing is specified when the arguments are arbitrary λ-terms.

3.3.3 Pairs. The encoding of pairs can be deduced from booleans. Namely, we can encode the pairing operator as

$$\text{pair} = \lambda xyb.\text{if}\, b\, x\, y$$

When applied to two terms t and u, it reduces to

$$\text{pair}\, t\, u \xrightarrow{*}_\beta \lambda b.\text{if}\, b\, t\, u$$

which can be thought of as an encoding of the pair $\langle t, u \rangle$. In order to recover the components of the pair, we can simply apply it to either T or F:

$$(\text{pair}\, t\, u)\, \mathsf{T} \xrightarrow{*}_\beta t \qquad\qquad (\text{pair}\, t\, u)\, \mathsf{F} \xrightarrow{*}_\beta u$$

We thus define the two projections as

$$\text{fst} = \lambda p.p\, \mathsf{T} \qquad\qquad \text{snd} = \lambda p.p\, \mathsf{F}$$

and we have, as expected

$$\text{fst}\,(\text{pair}\, t\, u) \xrightarrow{*}_\beta t \qquad\qquad \text{snd}\,(\text{pair}\, t\, u) \xrightarrow{*}_\beta u$$

More generally, n-uples can be encoded as

$$\text{uple}^n = \lambda x_1 \ldots x_n b.b\, x_1\, \ldots\, x_n$$

with the associated projections

$$\text{proj}_i^n = \lambda p.p\,(\lambda x_1 \ldots x_n.x_i)$$

and one checks that

$$\text{proj}_i^n\,(\text{uple}^n\, t_1\, \ldots\, t_n) \xrightarrow{*}_\beta t_i$$

3.3.4 Natural numbers. Given λ-terms f and x, and a natural number $n \in \mathbb{N}$, we write $f^n x$ for the λ-term $f(f(\ldots(fx)))$ with n occurrences of f:

$$f^0 x = x \qquad\qquad f^{n+1} x = f(f^n x)$$

The *n-th Church numeral* is the λ-term

$$\underline{n} = \lambda fx.f^n x = \lambda fx.f(f(\ldots(fx)))$$

In other words, the λ-term $\underline{n}$ is such that, when applied to arguments f and x, iterates n times the application of f to x. For low values of n, we have

$$\underline{0} = \lambda fx.x \quad\quad \underline{1} = \lambda fx.fx \quad\quad \underline{2} = \lambda fx.f(fx) \quad\quad \underline{3} = \lambda fx.f(f(fx)) \quad\quad \ldots$$

Successor. The successor function can be encoded as

$$\mathsf{succ} = \lambda nfx.f(nfx)$$

which applies f to $f^n\,x$. It behaves as expected since

$$
\begin{aligned}
\mathsf{succ}\,\underline{n} \;&\equiv\; (\lambda nfx.f(nfx))(\lambda fx.f^n x) \\
&\longrightarrow_\beta \lambda fx.f((\lambda fx.f^n x)fx) \\
&\longrightarrow_\beta \lambda fx.f((\lambda x.f^n x)x) \\
&\longrightarrow_\beta \lambda fx.f(f^n x) \\
&\equiv\; \lambda fx.f^{n+1}x \\
&\equiv\; \underline{n+1}
\end{aligned}
$$

Another natural possible definition of successor would be

$$\mathsf{succ} = \lambda nfx.nf(fx)$$

Arithmetic functions. The addition, multiplication and exponentiation can similarly be defined as

$$\mathsf{add} = \lambda mnfx.m\,\mathsf{succ}\,n \quad \mathsf{mul} = \lambda mnfx.m\,(\mathsf{add}\,n)\,0 \quad \mathsf{exp} = \lambda mn.n\,(\mathsf{mul}\,m)\,1$$

or, alternatively, as

$$\mathsf{add} = \lambda mnfx.mf(nfx) \qquad \mathsf{mul} = \lambda mnfx.m(nf)x \qquad \mathsf{exp} = \lambda mn.nm$$

It can be checked that addition is such that, for every $m, n \in \mathbb{N}$, we have $\mathsf{add}\,\underline{m}\,\underline{n} = \underline{m+n}$: it computes the function which to x associates f applied m times to f applied n times to x, i.e. $f^{m+n}\,x$. And similarly for other operations.

Comparisons. The test-if-zero function takes a natural number n as argument and returns the boolean true or false depending on whether n is 0 or not. It can be encoded as

$$\mathsf{iszero} = \lambda nxy.n(\lambda z.y)x$$

Given n, x and y, it applies the function $f = \lambda z.y$ n times to x: if the function is applied 0 times then x is returned, otherwise if the function is applied at least once then y is returned.

The predecessor function can also be encoded although it is more difficult (this is detailed below):

$$\mathsf{pred} = \lambda nfx.n(\lambda gh.h(gf))(\lambda y.x)(\lambda y.y)$$

It allows defining subtraction as

$$\mathsf{sub} = \lambda mn.n\,\mathsf{pred}\,m$$

where, by convention, the result of $m - n$ is 0 when $m < n$. From there, we can define comparisons of natural numbers such as the $\leqslant$ relation since $m \leqslant n$ is equivalent to $m - n = 0$:

$$\mathsf{leq} = \lambda mn.\mathsf{iszero}\,(\mathsf{sub}\,m\,n)$$

Exercise 3.3.4.1. The Ackermann function [Ack28] from pairs of natural numbers to natural numbers is the function A defined by

$$A(0, n) = n + 1$$
$$A(m + 1, 0) = A(m, 1)$$
$$A(m + 1, n + 1) = A(m, A(m + 1, n))$$

Show that, in λ-calculus, it can be implemented as

$$\mathsf{ack} = \lambda mn.m(\lambda f.\, n\, f(f\, \underline{1}))\, \mathsf{succ}$$

Predecessor. We are now going to see how we can implement the predecessor function mentioned above. Before going into that, let us see how we can implement the *Fibonacci sequence* f_n defined by $f_0 = 0$, $f_1 = 1$ and $f_{n+1} = f_n + f_{n-1}$. A naive implementation would be

```
let rec fib n =
  if n = 0 then 0
  else if n = 1 then 1
  else fib (n-1) + fib (n-2)
```

This function is highly inefficient because many computations are performed multiple times. For instance, to compute f_n, we compute both f_{n-1} and f_{n-2}, but the computation of f_{n-1} will require computing another time f_{n-2}, and so on. The usual strategy to improve that consists in computing two successive values (f_{n-1}, f_n) of the Fibonacci sequence at a time. Given such a pair, the next pair is computed by

$$(f_n, f_{n+1}) = (f_n, f_{n-1} + f_n)$$

We thus define the function

```
let fib_fun (q,p) = (p,p+q)
```

which computes the next pair depending on the current pair. If we iterate n times this function on the pair $(f_0, f_1) = (0, 1)$, we obtain the pair (f_n, f_{n+1}) and we can thus obtain the n-th term of the Fibonacci sequence by projecting to the first element:

```
let fib n = fst (iter n fib_fun (0,1))
```

where the function `iter` applies a function `f` `n` times to some element `x`:

```
let rec iter n f x =
  if n = 0 then x
  else f (iter (n-1) f x)
```

Now, suppose that we want to implement the predecessor function on natural numbers without using subtraction. Given $n \in \mathbb{N}$, there is one value for which we obviously know the predecessor: the predecessor of $n + 1$ is n. We will use this fact, and the above trick in order to remember the value for the previous predecessor, which is the $n - 1$ we are looking for! Let us write p_n for the

predecessor of n. We can compute the pair (p_n, p_{n+1}) of two successive values from the previous pair (p_{n-1}, p_n) by

$$(p_n, p_{n+1}) = (p_n, p_n + 1)$$

We thus define the function

```
let pred_fun (q,p) = (p,p+1)
```

If we iterate this function n times starting from the pair $(p_0, p_1) = (0, 0)$, we obtain the pair (p_n, p_{n+1}) and can thus compute p_n as its first component:

```
let pred n = fst (iter n pred_fun (0,0))
```

In λ-calculus, this translates as

$$\mathsf{pred} = \lambda n.\mathsf{fst}\,(n\,(\lambda x.\mathsf{pair}\,(\mathsf{snd}\,x)\,(\mathsf{succ}\,(\mathsf{snd}\,x)))\,(\mathsf{pair}\,\underline{0}\,\underline{0}))$$

The formula for predecessor provided above is a variant of this one.

3.3.5 Fixpoints. In order to define more elaborate functions on natural numbers such as the factorial, we need to have the possibility of defining functions recursively. This can be achieved in λ-calculus thanks to the so-called *fixpoint combinators*. In mathematics, a fixpoint of a function f is a value x such that $f(x) = x$. Note that such a value may or may not exist: for instance $f = x \mapsto x^2$ has 0 and 1 as fixpoints whereas $f = x \mapsto x + 1$ has no fixpoint.

Similarly, in λ-calculus a *fixpoint* for a term t is a term u such that

$$t\,u \mathrel{=\!=}_\beta u$$

A distinguishing feature of the λ-calculus is that

1. every term t admits a fixpoint,

2. this fixpoint can be computed within λ-calculus: there is a term Y such that $\mathsf{Y}\,t$ is a fixpoint of t:

$$t\,(\mathsf{Y}\,t) \mathrel{=\!=}_\beta \mathsf{Y}\,t$$

A term Y as above is called a *fixpoint combinator.*

Fixpoints in OCaml. Before giving a λ-term which is fixpoint operator, let us see how it can be implemented in OCaml and used to program recursive functions. In practice, we will look for a function Y such that

$$\mathsf{Y}\,t \longrightarrow_\beta t\,(\mathsf{Y}\,t)$$

Note that such a function is necessarily non-terminating since there is an infinite sequence of reductions

$$\mathsf{Y}\,t \longrightarrow_\beta t\,(\mathsf{Y}\,t) \longrightarrow_\beta t\,t\,(\mathsf{Y}\,t) \longrightarrow_\beta t\,t\,t\,(\mathsf{Y}\,t) \longrightarrow_\beta \ldots$$

but it might still be useful because since there might be other possible reductions reaching a normal form. Following the conventions, we will write `fix` instead of Y. A function which behaves as proposed is easily implemented:

```
let rec fix f = f (fix f)
```

Let us see how this can be used in order to implement the factorial function without explicitly resorting to recursion. The factorial function satisfies $0! = 1$ and $n! = n \times (n-1)!$ so that it can be implemented as

```
let rec fact n =
  if n = 0 then 1 else n * fact (n-1)
```

In order to implement it without using recursion, the trick is to first transform this function into one which takes, as first argument, a function `f` which is to be the factorial itself, and replace recursive calls by calls to this function:

```
let fact_fun f n =
  if n = 0 then 1 else n * f (n - 1)
```

We then expect the factorial function to be obtained as its fixpoint:

```
let fact = fix fact_fun
```

Namely, this function will reduce to `fact_fun (fix fact_fun)`, i.e. the above function where `f` was replaced by the function itself, as expected. However, if we try to define the function `fact` in this way, OCaml complains:

```
Stack overflow during evaluation (looping recursion?).
```

This is because OCaml always evaluates arguments first, so that it will fall into the infinite sequence of reductions mentioned above (the stack will grow at each recursive call and will exceed the maximal authorized value):

$$\text{fix fact\_fun} \longrightarrow_\beta \text{fact\_fun (fix fact\_fun)} \longrightarrow_\beta \ldots$$

The trick in order to avoid that in order to avoid that is to add an argument in the definition of `fix`:

```
let rec fix f x = f (fix f) x
```

and now the above definition of factorial computes as expected: this time, the argument `fix f` does not evaluate further because it is a function which is still expecting its second argument. It is interesting to note that the two definitions of `fix` (the looping one and the working one) are η-equivalent, see section 3.2.8, so that two η-equivalent terms can act differently depending on the properties we consider.

Fixpoints in λ-calculus. The above definition of `fix` does not easily translate to λ-calculus, because there is no simple way of defining recursive functions. A possible implementation of the fixpoint combinator can be obtained by a variant on the looping term Ω (see section 3.2.6). The *Curry fixpoint combinator* is

$$\mathsf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Namely, given a term t, we have

$$
\begin{aligned}
\mathsf{Y}\,t \;&\equiv\;\; (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\,t \\
&\longrightarrow_\beta (\lambda x.t(xx))(\lambda x.t(xx)) \\
&\longrightarrow_\beta t((\lambda x.t(xx))(\lambda x.t(xx))) \\
&{}_\beta\!\longleftarrow t\,(\mathsf{Y}\,t)
\end{aligned}
$$

which shows that we indeed have $\mathsf{Y}\,t \Longrightarrow_\beta t\,(\mathsf{Y}\,t)$, i.e. $\mathsf{Y}\,t$ is a fixpoint of t. Another possible fixpoint combinator is Turing's one defined as

$$\Theta = (\lambda fx.x(ffx))(\lambda fx.x(ffx))$$

which satisfies, for any term t,

$$\Theta\,t \stackrel{*}{\longrightarrow}_\beta t(\Theta\,t)$$

(we have here a proper sequence of β-reductions, as opposed to a mere β-equivalence for Curry's combinator).

The OCaml definition of the factorial

```
let fact = fix (fun f n -> if n = 0 then 1 else n * f (n - 1))
```

translates into λ-calculus as

$$\mathsf{fact} = \mathsf{Y}(\lambda fn.\mathsf{if}\,(\mathsf{iszero}\,n)\,\underline{1}\,(\mathsf{mul}\,n\,(f\,(\mathsf{pred}\,n))))$$

For instance, the factorial of 2 computes as

$$
\begin{aligned}
\mathsf{fact}\,\underline{2} \Longrightarrow\quad & (\mathsf{Y}F)\,\underline{2} \\
\Longrightarrow_\beta\ & F\,(\mathsf{Y}F)\,\underline{2} \\
\stackrel{*}{\longrightarrow}_\beta\ & \mathsf{if}\,(\mathsf{iszero}\,\underline{2})\,\underline{1}\,(\mathsf{mul}\,\underline{2}\,((\mathsf{Y}F)\,(\mathsf{pred}\,\underline{2}))) \\
\stackrel{*}{\longrightarrow}_\beta\ & \mathsf{if}\,\mathsf{false}\,\underline{1}\,(\mathsf{mul}\,\underline{2}\,((\mathsf{Y}F)\,(\mathsf{pred}\,\underline{2}))) \\
\stackrel{*}{\longrightarrow}_\beta\ & \mathsf{mul}\,\underline{2}\,((\mathsf{Y}F)\,(\mathsf{pred}\,\underline{2})) \\
\stackrel{*}{\longrightarrow}_\beta\ & \mathsf{mul}\,\underline{2}\,((\mathsf{Y}F)\,\underline{1}) \\
&\ \vdots \\
\stackrel{*}{\longrightarrow}_\beta\ & \mathsf{mul}\,\underline{2}\,(\mathsf{mul}\,\underline{1}\,\underline{1}) \\
\stackrel{*}{\longrightarrow}_\beta\ & \underline{2}
\end{aligned}
$$

Remark 3.3.5.1. Following Church's initial intuition when introducing the λ-calculus, we can think of λ-terms as describing sets, in the sense of set theory (see section 5.3). Namely, a set t can be thought of as a *predicate*, i.e. a function which takes an element u as argument and returns true or false depending on whether the element u belongs to t or not. Following this point of view, instead of writing $u \in t$, we write $t\,u$. Similarly, given a predicate t, the set $\{x \mid t\}$ is naturally written $\lambda x.t$:

set theory	λ-calculus
$u \in t$	$t\,u$
$\{x \mid t\}$	$\lambda x.t$

In this context, the paradoxical Russell set

$$r = \{x \mid \neg(x \in x)\}$$

of naive set theory, see section 5.3.1, is written as

$$r = \lambda x.\neg(xx)$$

This set has the property that $r \in r$ iff $\neg(r \in r)$, i.e.

$$rr = \neg(rr)$$

In other words rr is a fixpoint for $\neg$. Generalizing this to any f instead of $\neg$, we recover the definition of Y:

$$\mathsf{Y} = \lambda f.rr$$

with $r = \lambda x.f(xx)$. In this sense, the fixpoint combinator is the Russell paradox in disguise!

Church's combinator in OCaml. If we try to implement Church's combinator in OCaml:

```
let fix = fun f -> (fun x -> f (x x)) (fun x -> f (x x))
```

we get a typing error concerning the variable x. Namely, x is applied to something in the above expression, so it should be of type 'a -> 'b, but its argument is x itself, which imposes that we should have 'a = 'a -> 'b. The type of x should thus be the infinite type

```
... -> 'b -> 'b -> 'b -> 'b
```

which is not allowed by default. There are two ways around this.

The first one consists in using the -rectypes option of OCaml in order allow such types. If we use this function to define the factorial by

```
let fact = fix fact_fun
```

we get a stack overflow, meaning that the program is looping, which can be solved with an η-expansion (we have already seen this trick above). We can thus define instead

```
let fix = fun f -> (fun x y -> f (x x) y) (fun x y -> f (x x) y)
```

and now the definition of factorial works as expected.

The second one, if you do not want to use some exotic flag, consists in using a recursive type, which allows such recursions in types. Namely, we can define the type

```
type 'a t = Arr of ('a t -> 'a)
```

with which we can define the fixpoint operators as

```
let fix f =
  (fun x y -> f (arr x x) y) (Arr (fun x y -> f (arr x x) y))
```

where we use the shorthand

```
let arr (Arr f) = f
```

In the same spirit, the Turing fixpoint combinator can be implemented as

```
let turing =
  let t f x y = x (arr f f x) y in
  t (Arr t)
```

3.3.6 Turing completeness. The previous encodings of usual functions, should make it more or less clear that the λ-calculus is a full-fledged programming language. In particular, from the classical undecidability results [Tur37b] we can deduce:

Theorem 3.3.6.1 (Undecidability). The following problems are undecidable:

– whether two terms are β-equivalent,

– whether a term can β-reduce to a normal form.

In order to make this result more precise, we should encode Turing machines into λ-terms. Instead of doing this directly, we can rather encode recursive functions, which are already known to have the same expressiveness as Turing machines. The class of *recursive functions* is the smallest class of partially defined functions $f : \mathbb{N}^k \to \mathbb{N}$ for some $k \in \mathbb{N}$, which contains the zero constant function z, the successor function s and the projections p_i^k, for $k \in \mathbb{N}$ and $1 \leqslant i \leqslant k$:

$$\begin{array}{lll} \mathrm{z} : \mathbb{N}^0 \to \mathbb{N} & \mathrm{s} : \mathbb{N}^1 \to \mathbb{N} & \mathrm{p}_i^k : \qquad\quad \mathbb{N}^k \to \mathbb{N} \\ \qquad () \mapsto 0 & \qquad (n) \mapsto n+1 & \qquad\quad (n_1, \ldots, n_k) \mapsto n_i \end{array}$$

and is closed under

– composition: given recursive functions

$$f : \mathbb{N}^l \to \mathbb{N} \qquad \text{and} \qquad g_1, \ldots, g_l : \mathbb{N}^k \to \mathbb{N}$$

the function

$$\mathrm{comp}_{g_1,\ldots,g_l}^f : \qquad\quad \mathbb{N}^l \to \mathbb{N} \\ (n_1, \ldots, n_k) \mapsto f(g_1(n_1, \ldots, n_k), \ldots, g_l(n_1, \ldots, n_k))$$

is also recursive,

– primitive recursion: given recursive functions

$$f : \mathbb{N}^k \to \mathbb{N} \qquad \text{and} \qquad g : \mathbb{N}^{k+2} \to \mathbb{N}$$

the function

$$\mathrm{rec}_{f,g} : \qquad\qquad \mathbb{N}^{k+1} \to \mathbb{N} \\ (0, n_1, \ldots, n_k) \mapsto f(n_1, \ldots, n_k) \\ (n_0 + 1, n_1, \ldots, n_k) \mapsto g(\mathrm{rec}_{f,g}(n_0, n_1, \ldots, n_k), n_0, n_1, \ldots, n_k)$$

is also recursive,

– minimization: given a recursive function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ the function

$$\mathrm{min}_f : \mathbb{N}^k \to \mathbb{N}$$

which to $(n_1, \ldots, n_k) \in \mathbb{N}^k$ associates the smallest $n_0 \in \mathbb{N}$ such that $f(n_0, n_1, \ldots, n_k) = 0$ is also recursive.

The presence of minimization is the reason why we need to consider partially defined functions.

A function $f : \mathbb{N}^n \to \mathbb{N}$ is *definable* by a λ-term t when, for every tuple of natural numbers $(n_1, \ldots, n_k) \in \mathbb{N}^k$, we have

$$t \, \underline{n_1} \, \ldots \, \underline{n_k} \xrightarrow{\;*\;}_\beta \underline{f(n_1, \ldots, n_k)}$$

where $\underline{n}$ is the Church numeral associated to n.

Theorem 3.3.6.2 (Kleene). The functions definable in λ-calculus are precisely the recursive ones.

Proof. The terms constructed in section 3.3 easily allow to encode total recursive functions f as λ-terms $[\![f]\!]$: we define

$$[\![\mathrm{z}]\!] = \underline{0} = \lambda f x.x \qquad [\![\mathrm{s}]\!] = \mathsf{succ} = \lambda n f x.fnfx \qquad [\![\mathrm{p}^k_i]\!] = \lambda x_1 \ldots x_k.x_i$$

composition is given by

$$[\![\mathrm{comp}^f_{g_1, \ldots, g_l}]\!] = \lambda x_1 \ldots x_k.[\![f]\!]([\![g_1]\!]x_1 \ldots x_k) \ldots ([\![g_l]\!]x_1 \ldots x_k)$$

primitive recursion by

$$\begin{aligned}
[\![\mathrm{rec}_{f,g}]\!] = \mathsf{Y}(\lambda r x_0 x_1 \ldots x_k.\mathsf{if} \; (\mathsf{iszero} \, x_0) \\
([\![f]\!]x_1 \ldots x_k) \\
([\![g]\!](r(\mathsf{pred} \, x_0))(\mathsf{pred} \, x_0)x_1 \ldots x_k))
\end{aligned}$$

and minimization by

$$[\![\mathrm{min}_f]\!] = \mathsf{Y}(\lambda r x_0 x_1 \ldots x_k.\mathsf{if} \; (\mathsf{iszero} \, ([\![f]\!]x_0 x_1 \ldots x_k)) \, x_0 \, (r(\mathsf{succ} \, x_0)x_1 \ldots x_k)) \, \underline{0}$$

In order to handle general recursive functions, which might be partial, there is a subtlety with composition: if g is not defined on x, then $\mathrm{comp}^f_g(x)$ should not be defined, even if f is a constant function for instance, and this is not the case with the current encoding. This is easily overcome with the following construction: we write

$$t \downarrow u = u \, t \, (\lambda x.x)$$

for the term which should be read as "t provided u terminates". It can be checked that $t \downarrow u$ does not reduce to a normal form if u does not and $t \downarrow \underline{n} \xrightarrow{\;*\;}_\beta t$. We can now use this trick to correct the behavior of our encoding. For instance, the projection should be encoded as

$$[\![\mathrm{p}^k_i]\!] = \lambda x_1 \ldots x_k.(x_i \downarrow x_1 \downarrow \ldots \downarrow x_k)$$

For the converse property, i.e. the definable functions are recursive, we should encode λ-terms and their reduction into natural numbers, sometimes called *Gödel numbers*. This can be done, see [Bar84] (or if you are willing to accept that recursive functions are Turing-equivalent to usual programming languages, this amounts to showing that we can make a program which reduces λ-terms, which we can, see section 3.5).   □

3.3.7 Self-interpreting. We see that λ-calculus provides yet another model which is equivalent to Turing machines. This means that the functions we can compute in both model are the same, but not that they are equally simple to implement in both models. For instance, constructing a universal Turing machine is not an easy task: we have to decide on an encoding of the transitions on the tape and then build a Turing machine to use this encoding and the resulting machine is usually neither small nor particularly elegant.

In λ-calculus however, this is easy. For instance, we can encode a λ-term t as a λ-term $\ulcorner t \urcorner$, as follows. We first pick a fresh variable i (by fresh we mean here $i \notin \mathrm{FV}(t)$), replace every application $u\,v$ in t by $i\,u\,v$ and prepend λi to the resulting term. For instance, the term

$$t = \mathsf{succ}\,\underline{0} = (\lambda nfx.f(nfx))(\lambda fx.x)$$

is encoded as

$$\ulcorner t \urcorner = \lambda i.i(\lambda nfx.if(i(i\,n\,f)x))(\lambda fx.x)$$

Even though the original term t could reduce, the term $\ulcorner t \urcorner$ cannot (because of the manipulation we have performed on applications), and can thus be considered as a decent encoding of t. We can then define an *interpreter* as

$$\mathsf{int} = \lambda t.t\,(\lambda x.x)$$

This term has the property that, for every λ-term t, $\mathsf{int}\,\ulcorner t \urcorner$ β-reduces to the normal form of t. More details can be found in [Bar91, Mog92, Lyn17].

3.3.8 Adding constructors. Even though we have seen that all the usual constructions can be encoded in the λ-calculus, it is often convenient to add those as new explicit constructions to the calculus. For instance, products can be added to the λ-calculus by extending the syntax of λ-terms to

$$t, u ::= x \mid t\,u \mid \lambda x.t \mid \langle t, u \rangle \mid \pi_\mathrm{l} \mid \pi_\mathrm{r}$$

The new expressions are

– $\langle t, u \rangle$: the pair of two terms t and u,

– π_l and π_r: the left and right projections respectively.

The β-reduction also has to be extended in order to account for those. We add the two new reduction rules

$$\pi_\mathrm{l}\,\langle t, u \rangle \longrightarrow_\beta t \qquad\qquad \pi_\mathrm{r}\,\langle t, u \rangle \longrightarrow_\beta u$$

which express the fact that the left (resp. right) projection extracts the left (resp. right) component of a pair. Although most important properties (such as confluence) generalize to such variants of λ-calculus, we stick here to the plain one for simplicity. Some extensions are used and detailed in section 4.3.

3.4 Confluence of the λ-calculus

In order to be reasonably useful, the λ-calculus should be reasonably deterministic, i.e. we should be able to speak about "the" result of the evaluation (by

which we mean the β-reduction) of a λ-term. The first observation we already made is that, on given a term, multiple distinct reductions may be performed. For instance,

$$(\lambda xy.y)((\lambda a.a)(\lambda b.b))$$

$$y \qquad\qquad (\lambda xy.y)(\lambda b.b)$$

Another hope might be that if we reduce a term long enough, we will end up with a normal form (a term that cannot be reduced further), which can be considered as a result of the computation, and that if we perform two such reductions on a term, we will en up on the same normal form: the intermediate steps might not be the same, but in the end we always end up with the same result. For instance, on natural numbers, we can speak of 10 as the result of

$$(1 + 2) + (3 + 4)$$

because it does not depend on the intermediate steps used to compute it:

$$(1 + 2) + (3 + 4)$$

$$3 + (3 + 4) \qquad\qquad (1 + 2) + 7$$

$$3 + 7$$

$$\downarrow$$

$$10$$

However, in the case of λ-calculus, this hope is vain because we have seen that some terms might lead to infinite sequence of β-reductions, thus never reaching a normal form.

3.4.1 Confluence. The property which turns out to be satisfied in the case of λ-calculus is called *confluence*: it states that if starting from a term t reduces in many steps to a term u_1 and also to a term u_2, then there exists a term v such that both u_1 and u_2 reduce in many steps to v:

$$t$$

$$u_1 \qquad\qquad u_2$$

$$v$$

In other words, computation starting from a term t might lead to different intermediate results, but there is always a way for those results to converge to a common term.

Note that this result would not be valid if we required to have exactly one reduction step each time. For instance, we need two reductions to complete the

following square on the right:

$$(\lambda yx.xyy)(\mathsf{I}\,\mathsf{I})$$

$$(\lambda yx.xyy)\,\mathsf{I} \qquad\qquad \lambda x.x(\mathsf{I}\,\mathsf{I})(\mathsf{I}\,\mathsf{I})$$

$$\lambda x.x(\mathsf{I}\,\mathsf{I})\mathsf{I}$$

$$\lambda x.x\,\mathsf{I}\,\mathsf{I}$$
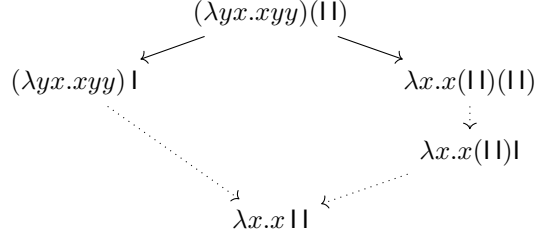
where $\mathsf{I} = \lambda x.x$ is the identity. The easiest way to prove this confluence result first requires to introduce a variant of the β-reduction.

3.4.2 The parallel β-reduction. The *parallel β-reduction* $\longrightarrow\!\!\!\!\rightarrow$ is the smallest relation on λ-terms such that

$$\frac{}{x \longrightarrow\!\!\!\!\rightarrow x}\ (\beta_{\mathsf{x}}^{\|}) \qquad\qquad \frac{t \longrightarrow\!\!\!\!\rightarrow t' \qquad u \longrightarrow\!\!\!\!\rightarrow u'}{(\lambda x.t)u \longrightarrow\!\!\!\!\rightarrow t'[u'/x]}\ (\beta_{\mathsf{s}}^{\|})$$

$$\frac{t \longrightarrow\!\!\!\!\rightarrow t' \qquad u \longrightarrow\!\!\!\!\rightarrow u'}{t\,u \longrightarrow\!\!\!\!\rightarrow t'\,u'}\ (\beta_{\mathsf{a}}^{\|}) \qquad\qquad \frac{t \longrightarrow\!\!\!\!\rightarrow t'}{\lambda x.t \longrightarrow\!\!\!\!\rightarrow \lambda x.t'}\ (\beta_{\lambda}^{\|})$$

As usual, we write $\overset{*}{\longrightarrow\!\!\!\!\rightarrow}$ for the reflexive and transitive closure of the relation $\longrightarrow\!\!\!\!\rightarrow$.

Informally, $t \longrightarrow\!\!\!\!\rightarrow u$ means that u is obtained from t by reducing in one step many of the β-redexes present in t at once. For instance, we have

$$(\lambda xy.\mathsf{I}\,x\,y)(\mathsf{I}\,\mathsf{I}) \longrightarrow\!\!\!\!\rightarrow \lambda y.\mathsf{I}y \longrightarrow\!\!\!\!\rightarrow \lambda y.y$$

where the first step intuitively corresponds to simultaneously performing the three β-reductions

$$(\lambda xy.\mathsf{I}\,x\,y)(\mathsf{I}\,\mathsf{I}) \longrightarrow_{\beta} \lambda y.\mathsf{I}\,(\mathsf{I}\,\mathsf{I})\,y \qquad \mathsf{I}\,x \longrightarrow_{\beta} x \qquad \mathsf{I}\,\mathsf{I} \longrightarrow_{\beta} \mathsf{I}$$

As for usual β-reduction, the parallel β-reduction might create some β-redexes which were not present in the original term, and could thus not be reduced at first. For this reason, even though we can reduce in multiple places at once, we cannot perform a parallel β-reduction step directly from the term on the left to the term on the right in the above example.

In parallel β-reduction, we are allowed not to perform all the available β-reduction steps. In particular, we may perform none:

Lemma 3.4.2.1. For every λ-term t, we have $t \longrightarrow\!\!\!\!\rightarrow t$.

Proof. By induction on the term t. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

3.4.3 Properties of the parallel β-reduction. We now study some properties of the parallel β-reduction. Since it corresponds to performing β-reduction steps in parallel, the relations $\overset{*}{\longrightarrow\!\!\!\!\rightarrow}$ and $\overset{*}{\longrightarrow}_{\beta}$ coincide: we can simulate parallel β-reduction with β-reduction and conversely. Moreover, we will see that parallel β-reduction is easily shown to be confluent, from which we will be able to deduce the confluence of β-reduction.

First, any β-reduction step can be simulated by a parallel reduction step:

Lemma 3.4.3.1. If $t \longrightarrow_\beta u$ then $t \longrightarrow\!\!\!\!\twoheadrightarrow u$.

Proof. By induction on the derivation of $t \longrightarrow_\beta u$. $\square$

Conversely, any parallel β-reduction step can be simulated by multiple β-reduction steps:

Lemma 3.4.3.2. If $t \longrightarrow\!\!\!\!\twoheadrightarrow u$ then $t \xrightarrow{*}_\beta u$.

Proof. By induction on the derivation of $t \longrightarrow\!\!\!\!\twoheadrightarrow u$. $\square$

From this, we immediately deduce that the reflexive and transitive closure of the two relations coincide:

Lemma 3.4.3.3. We have $t \xrightarrow{*}\!\!\!\twoheadrightarrow u$ if and only if $t \xrightarrow{*}_\beta u$.

Proof. If $t \xrightarrow{*}\!\!\!\twoheadrightarrow u$, this means that we have a sequence of parallel reduction steps

$$t = t_0 \longrightarrow\!\!\!\!\twoheadrightarrow t_1 \longrightarrow\!\!\!\!\twoheadrightarrow t_2 \longrightarrow\!\!\!\!\twoheadrightarrow \ldots \longrightarrow\!\!\!\!\twoheadrightarrow t_n = u$$

Therefore, by lemma 3.4.3.2,

$$t = t_0 \xrightarrow{*}_\beta t_1 \xrightarrow{*}_\beta t_2 \xrightarrow{*}_\beta \ldots \xrightarrow{*}_\beta t_n = u$$

and thus $t \xrightarrow{*}_\beta u$. Conversely, if $t \xrightarrow{*}_\beta u$, this means that we have a sequence of β-reduction steps

$$t = t_0 \longrightarrow_\beta t_1 \longrightarrow_\beta t_2 \longrightarrow_\beta \ldots \longrightarrow_\beta t_n = u$$

Therefore, by lemma 3.4.3.2,

$$t = t_0 \longrightarrow\!\!\!\!\twoheadrightarrow t_1 \longrightarrow\!\!\!\!\twoheadrightarrow t_2 \longrightarrow\!\!\!\!\twoheadrightarrow \ldots \longrightarrow\!\!\!\!\twoheadrightarrow t_n = u$$

and thus $t \xrightarrow{*}\!\!\!\twoheadrightarrow u$. $\square$

Next, the parallel β-reduction is compatible with substitution.

Lemma 3.4.3.4. If $t \longrightarrow\!\!\!\!\twoheadrightarrow t'$ and $u \longrightarrow\!\!\!\!\twoheadrightarrow u'$ then $t[u/x] \longrightarrow\!\!\!\!\twoheadrightarrow t'[u'/x]$.

Proof. By induction on the derivation of $t \longrightarrow\!\!\!\!\twoheadrightarrow t'$.

– If the last rule is

$$\frac{}{y \longrightarrow\!\!\!\!\twoheadrightarrow y} \; (\beta_{\mathsf{x}}^{\|})$$

then $t = y = t'$ and we conclude with

$$y[u/x] = y \longrightarrow\!\!\!\!\twoheadrightarrow y = y[u/x]$$

or

$$x[u/x] = u \longrightarrow\!\!\!\!\twoheadrightarrow u' = x[u/x]$$

depending on whether $y \neq x$ or $y = x$.

– If the last rule is

$$\frac{t_1 \longrightarrow t_1' \qquad t_2 \longrightarrow t_2'}{(\lambda y.t_1)\, t_2 \longrightarrow t_1'[t_2'/y]} \ (\beta_{\mathsf{s}}^{\|})$$

with $y \neq x$, then, by induction hypothesis, we have

$$t_1[u/x] \longrightarrow t_1'[u'/x] \qquad\qquad t_2[u/x] \longrightarrow t_2'[u'/x]$$

and thus, by $(\beta_{\mathsf{s}}^{\|})$,

$$(\lambda y.t_1[u/x])\, (t_2[u/x]) \longrightarrow t_1'[u'/x][t_2'[u'/x]/y]$$

which can be rewritten as

$$((\lambda y.t_1)\, t_2)[u/x] \longrightarrow t_1'[t_2'/y][u'/x]$$

– If the last rule is

$$\frac{t_1 \longrightarrow t_1' \qquad t_2 \longrightarrow t_2'}{t_1\, t_2 \longrightarrow t_1'\, t_2'} \ (\beta_{\mathsf{a}}^{\|})$$

then, by induction hypothesis, we have

$$t_1[u/x] \longrightarrow t_1'[u'/x] \qquad\qquad t_2[u/x] \longrightarrow t_2'[u'/x]$$

and thus, by $(\beta_{\mathsf{a}}^{\|})$,

$$(t_1[u/x])\, (t_2[u/x]) \longrightarrow (t_1'[u'/x])\, (t_2'[u'/x])$$

in other words

$$(t_1\, t_2)[u/x] \longrightarrow (t_1'\, t_2')[u'/x]$$

– If the last rule is

$$\frac{t_1 \longrightarrow t_1'}{\lambda y.t_1 \longrightarrow \lambda y.t_1'} \ (\beta_{\lambda}^{\|})$$

the by induction hypothesis we have

$$t_1[u/x] \longrightarrow t_1'[u'/x]$$

and thus, by $(\beta_{\lambda}^{\|})$,

$$(\lambda y.t_1)[u/x] = \lambda y.t_1[u/x] \longrightarrow \lambda y.t_1'[u'/x] = (\lambda y.t_1')[u'/x]$$

and we are done. $\qquad\square$

We can use this lemma to show that the β-reduction satisfies a variant of the confluence property called the *diamond property*, or *local confluence*:

Lemma 3.4.3.5 (Diamond property). Suppose that $t \longrightarrow u$ and $t \longrightarrow u'$. Then there exists v such that $u \longrightarrow v$ and $u' \longrightarrow v$:
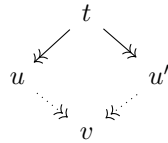
Proof. Suppose that $t \longrightarrow\!\!\!\!\!\rightarrow u$ and $t \longrightarrow\!\!\!\!\!\rightarrow u'$. We show the result by induction on the derivation of $t \longrightarrow\!\!\!\!\!\rightarrow u$.

– If the last rule of the derivation of $t \longrightarrow\!\!\!\!\!\rightarrow u$ is

$$\frac{}{x \longrightarrow\!\!\!\!\!\rightarrow x} \; (\beta_{\mathsf{x}}^{\|})$$

then $t = x = u$ and, by lemma 3.4.2.1, we have $u' \longrightarrow\!\!\!\!\!\rightarrow u'$:
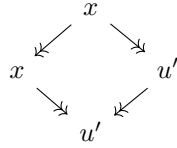
– If the last rule of the derivation of $t \longrightarrow\!\!\!\!\!\rightarrow u$ is

$$\frac{t_1 \longrightarrow\!\!\!\!\!\rightarrow u_1 \qquad t_2 \longrightarrow\!\!\!\!\!\rightarrow u_2}{(\lambda x.t_1)t_2 \longrightarrow\!\!\!\!\!\rightarrow u_1[u_2/x]} \; (\beta_{\mathsf{s}}^{\|})$$

we have two possible cases depending on the derivation of $t \longrightarrow\!\!\!\!\!\rightarrow u'$.

– If the last rule of the derivation of $t \longrightarrow\!\!\!\!\!\rightarrow u'$ is

$$\frac{t_1 \longrightarrow\!\!\!\!\!\rightarrow u'_1 \qquad t_2 \longrightarrow\!\!\!\!\!\rightarrow u'_2}{(\lambda x.t_1)t_2 \longrightarrow\!\!\!\!\!\rightarrow u'_1[u'_2/x]} \; (\beta_{\mathsf{s}}^{\|})$$

then, by induction hypothesis, there exists a term v_i such that $u_i \longrightarrow\!\!\!\!\!\rightarrow v_i$ and $u'_i \longrightarrow\!\!\!\!\!\rightarrow v_i$, for $i = 1$ or $i = 2$:
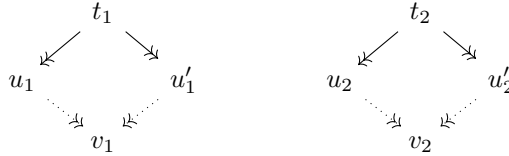
Therefore, we have both

$$u_1[u_2/x] \longrightarrow\!\!\!\!\!\rightarrow v_1[v_2/x] \qquad \text{and} \qquad u'_1[u'_2/x] \longrightarrow\!\!\!\!\!\rightarrow v_1[v_2/x]$$

by lemma 3.4.3.4 and we can conclude:
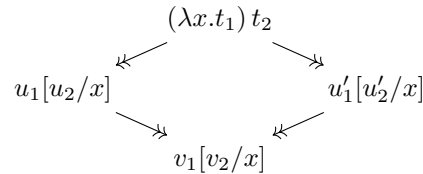
– If the last rule of the derivation of $t \longrightarrow\!\!\!\!\!\rightarrow u'$ is

$$\frac{\lambda x.t_1 \longrightarrow\!\!\!\!\!\rightarrow t'_1 \qquad t_2 \longrightarrow\!\!\!\!\!\rightarrow u'_2}{(\lambda x.t_1)\,t_2 \longrightarrow\!\!\!\!\!\rightarrow t'_1\,u'_2} \; (\beta_{\mathsf{a}}^{\|})$$

then the last rule of the derivation of $\lambda x.t_1 \longrightarrow t_1'$ is necessarily of the form

$$\frac{t_1 \longrightarrow u_1'}{\lambda x.t_1 \longrightarrow \lambda x.u_1'} \; (\beta_\lambda^\|)$$

with $t_1' = \lambda x.u_1'$. By induction hypothesis, we have the existence of the dotted reductions
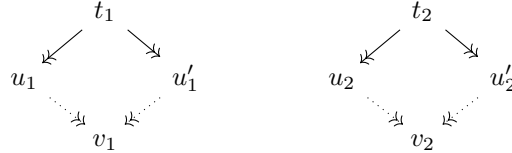
We thus have

$$u_1[u_2/x] \longrightarrow v_1[v_2/x]$$

by lemma 3.4.3.4 and

$$(\lambda x.u_1')\, u_2' \longrightarrow v_1[v_2/x]$$

by $(\beta_s^\|)$, from which we conclude:
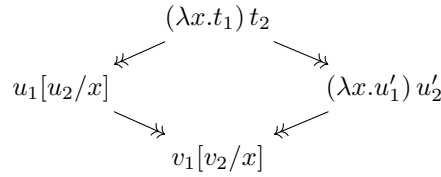
– If the last rule of the derivation of $t \longrightarrow u$ is

$$\frac{t_1 \longrightarrow u_1 \qquad t_2 \longrightarrow u_2}{t_1\, t_2 \longrightarrow u_1\, u_2} \; (\beta_a^\|)$$

the derivation of $t \longrightarrow u'$ ends either with $(\beta_s^\|)$ or $(\beta_a^\|)$ and both cases are handled similarly as above.

– If the last rule of the derivation of $t \longrightarrow u$ is

$$\frac{t_1 \longrightarrow u_1}{\lambda x.t_1 \longrightarrow \lambda x.u_1} \; (\beta_\lambda^\|)$$

we can reason similarly as above. $\qquad\square$

From this follows easily the confluence property of the relation $\longrightarrow$ in two steps:

Lemma 3.4.3.6. Suppose that $t \longrightarrow u$ and $t \stackrel{*}{\longrightarrow} u'$. Then there exists v such that $u \stackrel{*}{\longrightarrow} v$ and $u' \longrightarrow v$:
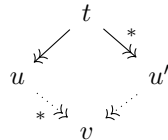
Proof. By induction on the length of the upper-right reduction $t \xrightarrow{*} u'$, using lemma 3.4.3.5. $\square$

Theorem 3.4.3.7 (Confluence). Suppose that $t \xrightarrow{*} u$ and $t \xrightarrow{*} u'$. Then there exists v such that $u \xrightarrow{*} v$ and $u' \xrightarrow{*} v$:

$$
\begin{array}{ccc}
 & t & \\
{}^{*}\swarrow & & \searrow^{*} \\
u & & u' \\
{}_{*}\searrow & & \swarrow_{*} \\
 & v &
\end{array}
$$
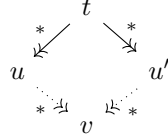
Proof. By induction on the length of the upper-left reduction $t \xrightarrow{*} u$, using lemma 3.4.3.6. $\square$

3.4.4 Confluence and the Church-Rosser theorem. As a consequence of the above lemmas, we can finally deduce the confluence property of λ-calculus, first proved by Church and Rosser [CR36], the proof presented here being due to Tait and Martin-Löf:

Theorem 3.4.4.1 (Confluence). The β-reduction is confluent: if $t \xrightarrow{*}_{\beta} u_1$ and $t \xrightarrow{*}_{\beta} u_2$ then there exists v such that $u_1 \xrightarrow{*}_{\beta} v$ and $u_2 \xrightarrow{*}_{\beta} v$:

$$
\begin{array}{ccc}
 & t & \\
{}^{*}\swarrow & & \searrow^{*} \\
u & & u' \\
{}_{*}\searrow & & \swarrow_{*} \\
 & v &
\end{array}
$$
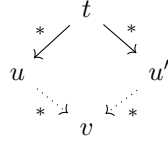
Proof. Suppose that $t \xrightarrow{*}_{\beta} u_1$ and $t \xrightarrow{*}_{\beta} u_2$. By lemma 3.4.3.3, we have $t \xrightarrow{*} u_1$ and $t \xrightarrow{*} u_2$. From theorem 3.4.3.7, we deduce the existence of v such that $u_1 \xrightarrow{*} v$ and $u_2 \xrightarrow{*} v$ and, by lemma 3.4.3.3 again, we have $u_1 \xrightarrow{*}_{\beta} v$ and $u_2 \xrightarrow{*}_{\beta} v$. $\square$

This implies the following theorem, sometimes called the *Church-Rosser property* of λ-calculus:

Theorem 3.4.4.2 (Church-Rosser). Given two terms t and u such that $t =\!=_{\beta} u$, there exists a term v such that $t \xrightarrow{*}_{\beta} v$ and $u \xrightarrow{*}_{\beta} v$:

$$
\begin{array}{ccc}
t & \xleftrightarrow{\quad * \quad} & u \\
{}_{*}\searrow & & \swarrow_{*} \\
 & v &
\end{array}
$$
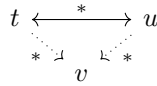
Proof. By definition of β-equivalence, see section 3.2.7, there is $n \in \mathbb{N}$ and terms t_i for $0 \leqslant i \leqslant 2n$ such that

$$
t = t_0 {}_{\beta}\!\xleftarrow{*} t_1 \xrightarrow{*}_{\beta} t_2 {}_{\beta}\!\xleftarrow{*} t_3 \xrightarrow{*}_{\beta} \ldots {}_{\beta}\!\xleftarrow{*} t_{2n-1} \xrightarrow{*}_{\beta} t_{2n} = u
$$

We show the result by induction on n. For $n = 0$, the result is obvious. Otherwise, we can complete the diagram as follows:
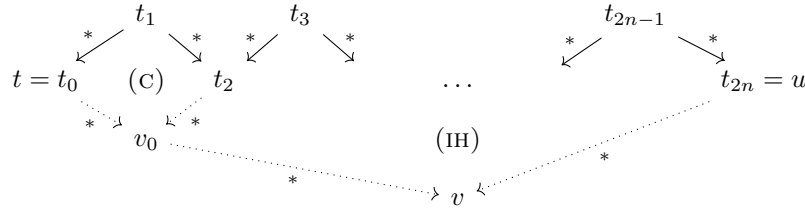
where (C) is obtained by theorem 3.4.4.1 and (IH) by induction hypothesis. □

One of the most important consequences is that a λ-term cannot reduce to two distinct normal forms: if the computation terminates then its result is uniquely defined.

Proposition 3.4.4.3. If t and u are two β-equivalent terms in normal forms then $t = u$.

Proof. By theorem 3.4.4.2, there exists v such that $t \xrightarrow{*}_\beta v$ and $u \xrightarrow{*}_\beta v$. Since t and u are normal forms, they cannot reduce and thus $t = v = u$. □

Another byproduct is the so-called *consistency* of λ-calculus which states that the β-equivalence relation does not identify all terms:

Theorem 3.4.4.4 (Consistency). There are terms which are not β-equivalent.

Proof. The terms $\lambda xy.x$ and $\lambda xy.y$ are normal forms. If they were equivalent they would be equal by previous proposition, which is not the case. □

3.5 Implementing reduction

3.5.1 Reduction strategies. We have seen that a λ-term can reduce in many ways, but in practice people implement a particular deterministic way of choosing reductions to perform: this is called a *reduction strategy*. This is the case for OCaml and is easily observed by inserting prints. For instance, the program

```
let p = print_endline
let _ = (p "a"; (fun x y -> p "b"; x + y)) (p "c"; 2) (p "d"; 3)
```

will always print `dcab` in the toplevel. We shall now try to look at the options we have here, in order to chose a strategy. A first question we have to answer is: should we reduce functions or arguments first? Namely, consider a term of the form $(\lambda x.t)u$ such that u reduces to u', we have two possible ways of reducing it:

$$t[u/x]_\beta \longleftarrow (\lambda x.t)u \longrightarrow_\beta (\lambda x.t)u'$$

which correspond to reducing functions or arguments first, giving rise to strategies which are respectively called *call-by-name* and *call-by-value*. The call-by-value has a tendency to be more efficient: even if the argument is used multiple times in the function, we reduce it only once beforehand, whereas the call-by-name strategy reduces it each time it is used. For instance, if $u \xrightarrow{*}_\beta \hat{u}$, where $\hat{u}$ is a normal form, we have the following sequences of reductions:

– in call-by-value: $(\lambda x.fxx)u \xrightarrow{*}_\beta (\lambda x.fxx)\hat{u} \longrightarrow_\beta f\hat{u}\hat{u}$,

– in call-by-name: $(\lambda x.fxx)u \longrightarrow_\beta fuu \xrightarrow{*}_\beta f\hat{u}u \xrightarrow{*}_\beta f\hat{u}\hat{u}$.

The function $\lambda x.fxx$ uses its argument twice and therefore we have to reduce u twice in the second case compared to only once in the first (and this can make a huge difference if the argument is used much more than twice or if the reduction of u requires many steps). However, there is a case where the call-by-value strategy is inefficient: when the argument is not used in the function. Namely, we always reduce the argument, even if it is not used afterwards. For instance, we have the following sequences of reductions:

– in call-by-value: $(\lambda x.y)u \xrightarrow{*}_\beta (\lambda x.y)\hat{u} \longrightarrow_\beta y$

– in call-by-name: $(\lambda x.y)u \longrightarrow_\beta y$

We have already observed in section 3.2.3 that β-reduction can duplicate and erase β-redexes: the call-by-value strategy is optimized for duplication and the call-by-name strategy is optimized for erasure. In practice, people often write programs where they use a result multiple times and rarely discard the result of computations, so that call-by-value strategies are generally implemented (this is for instance the case in OCaml). However, for theoretical purposes call-by-value strategies can be a problem: it might happen that a term has a normal form and that this strategy does not find it. Namely, consider the term

$$(\lambda x.y)\Omega$$

A call-by-value strategy will first try to compute the normal form for Ω and thus loop, whereas a call-by-name strategy will directly reduce it to y. A strategy is called *normalizing* when it will reach a normal form whenever a term has one: we have seen that call-by-value does not have this property.

Orders on redexes. In more precise terms, to define a reduction strategy, we have to chose the order in which we will reduce the redexes. Two partial orders can be defined on redexes:

– the *imbrication order*: a redex is *inside* another redex when it is a subterm of it, i.e. the redexes of t or of u are inside the redex

$$(\lambda x.t)u$$

– the *horizontal order*: in a subterm

$$t\,u$$

every redex in t is *on the left* of every redex in u.

Any two redexes in a term can be compared with one of those orders; a strategy can thus be specified by which redexes it favors with respect to each of these orders:

– a strategy is *innermost* (resp. *outermost*) when it begins with redexes which are the most inside (resp. outside),

– a strategy is *left* (resp. *right*) when it begins with redexes which are the most on the left (resp. right).

For instance, the above examples illustrate the fact that the call-by-value and call-by-name strategies are respectively innermost and outermost.

Partial evaluation. Another possibility which is generally offered when defining a reduction strategy is to allow not reducing some terms which are not in normal form. These terms can be thought of as "incomplete" and we are waiting for some more information (e.g. an argument or the value of a free variable) in order to further reduce them. Two such families are mainly considered:

– a strategy is *weak* when it does not reduce abstractions: a term of the form $\lambda x.t$ will never be reduced, even if the term t contains redexes,

– a left strategy is *head* when it does not reduce variables applied to terms: a term of the form $x\,t_1\ldots t_n$ will never be reduced, even if some term t_i contains redexes.

A strategy is *full* when it is neither weak nor head.

The reason for considering weak strategies is that a function is usually thought of as describing the actions to perform once arguments are given and it is therefore natural to delay their execution until we actually know those arguments. For instance, the strategy implemented in OCaml is weak: if it was not the case then the program

```
let f n =
  print_endline "Incrementing!";
  n+1
```

would always print the message exactly once, even if the function is never called, whereas we expect that the message is printed each time the function is called (which is the case with a weak evaluation strategy). In pure λ-calculus, there is no printing but one thing is easily observed: non-termination. For instance, we want to be able to define a function which loops or not depending on a boolean as follows:

$$\lambda b.\text{if } b\,\Omega\,\mathsf{I}$$

This function takes a boolean as argument: if it is true it will return the term Ω whose evaluation is going to loop, otherwise it returns the identity. If we evaluate it with a weak strategy it will behave as expected, whereas if we use a non-weak one, we might evaluate the body of the abstraction and thus loop when reducing Ω, even if we give false as argument to the function.

Head reductions mostly make sense for strategies like call-by-name: in a term $(\lambda x.t)u$, we reduce to $t[u/x]$ even if u is not in normal form because we want to delay the evaluation of u until it is actually used. Now, in a term $x\,u$, it might be the case that the free variable x will be replaced by an abstraction later on, and we want therefore to delay the evaluation of u until this is the case.

A term which cannot be reduced by a weak or head strategy is not necessarily in normal form in the usual sense. Recall from proposition 3.2.6.1 that λ-terms in normal form can be described by the following grammar:

$$v ::= \lambda x.v \mid x\,v_1\ldots v_n$$

where v and v_i are normal forms. A term is

– a *weak normal form* when it is generated by

$$v ::= \lambda x.t \mid x\,v_1\ldots v_n$$

where t is a term and the v_i are weak normal forms,

– a *head normal form* when it is generated by

$$v ::= \lambda x.v \mid x\, t_1 \ldots t_n$$

where v is a head normal form and the t_i are terms,

– a *weak head normal form* when it is generated by

$$v ::= \lambda x.t \mid x\, t_1 \ldots t_n$$

where t and the t_i are terms.

The terms which cannot be reduced in a weak (resp. head, resp. weak head) strategy are precisely weak (resp. head, resp. weak head) normal forms. Weak normal forms coincide with normal forms for terms which are not abstractions (resp. closed terms): they do the job if we are mostly interested in those terms, which we usually are. However, there are function which are weak (resp. head) normal forms such as $\lambda x.\mathsf{II}$ (resp. $x\,(\mathsf{II})$) and are not normal forms, so that a weak (resp. head) strategy is never normalizing.

Summary of strategies. We will detail below four reduction strategies, whose main properties are summarized below. Those strategies are the most well-known and used ones, but other variants could of course be considered:

	left.	inner.	weak	head	norm.
AO	✓	✓			
CBV	✓	✓	✓		
NO	✓				✓
CBN	✓		✓	✓	

The columns respectively indicate whether the strategy is leftmost (or rightmost), innermost (or outermost), weak, head and normalizing.

Implementing λ-terms. In order to illustrate implementations of reduction strategies in OCaml, we will encode λ-terms using the type

```
type term =
  | Var of var
  | App of term * term
  | Abs of var * term
```

where `var` is an arbitrary type for identifying variables (in practice, we would choose `int` or maybe `string`). We will also need a substitution function, such that `subst x t u` computes the term `u` where all occurrences of the variable `x` have been replaced by the term `t`:

```
let rec subst x t = function
  | Var y       -> if x = y then t else Var y
  | App (u, v)  -> App (subst x t u, subst x t v)
  | Abs (y', u) ->
    let y = fresh () in
    let u = subst y' (Var y) u in
    Abs (y, subst x t u)
```

In order to avoid name captures, we always refresh the names of the abstracted variables when substituting under an abstraction (this is correct, but quite inefficient): in order to do so we use a function `fresh` which generates a new variable name each time it is called, e.g. using an internal counter incremented at each call. For each of the considered strategies below, we will define a function `reduce` which performs multiple β-reduction steps in the order specified by the strategy.

Call-by-value. The *call-by-value* strategy (CBV) is by far the most common; it is the one used by OCaml for instance. Its name comes from the fact that it computes the value of the argument of a function before applying the function to the argument. It is defined as the weak leftmost innermost strategy. This means that, given an application $t\,u$,

1. we evaluate t until we obtain a term of the form $\lambda x.t'$ (where t' is not necessarily a normal form),

2. we then evaluate the argument u to a weak normal form $\hat{u}$,

3. we then evaluate $t'[\hat{u}/x]$.

The reduction function associated to this strategy can be implemented as follows:

```
let rec reduce = function
  | Var x     -> Var x
  | Abs (x, t) -> Abs (x, t)
  | App (t, u) ->
    match reduce t with
    | Abs (x, t') -> subst x (reduce u) t'
    | t           -> App (t, reduce u)
```

In the case `App (t, u)` it can be observed that both terms `t` and `u` are always reduced, so that taking the rightmost variant of the strategy has little effect. Since it is a weak strategy, it is not normalizing, and normal forms for this strategy will be weak normal forms. The above function does not directly compute the weak normal form: it has to be iterated. For instance, applying it to $(\lambda x.xy)(\lambda x.x)$ will result in $(\lambda x.x)y$, which further reduces to y.

Applicative order. The *applicative order* strategy (AO) is the leftmost innermost strategy, i.e. the variant of call-by-name where we are allowed to reduce under abstractions.

```
let rec reduce = function
  | Var x     -> Var x
  | Abs (x, t) -> Abs (x, reduce t)
  | App (t, u) ->
    match reduce t with
    | Abs (x, t') -> subst x (reduce u) t'
    | t           -> App (t, reduce u)
```

Normal forms are normal forms in the usual sense. As illustrated above, by the term $(\lambda x.y)\Omega$, this strategy might not terminate even though the term has a normal form, i.e. the strategy is not normalizing.

Call-by-name. The *call-by-name* strategy (CBN) is the weak head leftmost outermost strategy. Here, arguments are computed at each use and not once for all as in call-by-value strategy. An implementation of the corresponding reduction is

```
let rec reduce = function
  | Var x      -> Var x
  | Abs (x, t) -> Abs (x, t)
  | App (t, u) ->
    match reduce t with
    | Abs (x, t') -> subst x u t'
    | t           -> App (t, u)
```

Iterating this function computes the weak head normal form for a term, which is the appropriate notion of normal form for the strategy. This strategy being weak and head, it is not normalizing. However, it can be shown that if a term has a weak head normal form, this strategy will compute it (this can be obtained as a variant of the corresponding result for the normal order, see below).

Normal order. The *normal order* strategy (NO) is the leftmost outermost strategy. An implementation is

```
let rec reduce = function
  | Var x      -> Var x
  | Abs (x, t) -> Abs (x, reduce t)
  | App (t, u) ->
    match reduce_cbn t with
    | Abs (x, t') -> subst x u t'
    | t           -> App (reduce t, reduce u)
```

where `reduce_cbn` is the above call-by-name reduction strategy. Normal forms for this strategy are normal forms in the usual sense, and this strategy is actually normalizing: it can be shown that if there is a way to reduce a λ-term to a normal form then this strategy will find it, thus its name: this is a consequence of the so-called *standardization theorem* in λ-calculus [Bar84, chapters 12 and 13], [SU06, theorem 1.5.8].

Normalization. As indicated earlier, the `reduce` functions perform multiple reduction steps in the order specified by the strategy, so that iterating it reduces the term to its normal form following the strategy. A term can thus be normalized according to one of the strategies using the following function:

```
let rec normalize t =
  let u = reduce t in
  if t = u then t else normalize u
```

3.5.2 Normalization by evaluation. The implementations provided in section 3.5.1 are not really efficient because of one small reason: the substitution function is not implemented efficiently. Doing this efficiently, while properly taking bound variables in account, is actually quite difficult, see section 3.6.2. When using a functional language such as OCaml, the compiler already has

support for that and we can use the reduction of the host language in order to implement the β-reduction and compute normal forms. This is called *normalization by evaluation*: we implement the normalization function using the evaluation of the language. We shall now see how to perform that in practice.

Evaluation. We begin by describing our λ-terms as usual:

```
type term =
  | Var of string
  | Abs of string * term
  | App of term * term
```

For convenience, variable names are described by strings. For instance, we can define the looping λ-term $\Omega = (\lambda x.xx)(\lambda x.xx)$ by

```
let omega =
  let o = Abs ("x", App (Var "x", Var "x")) in
  App (o, o)
```

We are going to evaluate those terms to normal forms, which we call here *values*. We know from proposition 3.2.6.1 that those normal forms can be characterized as the λ-terms v generated by the grammar

$$v ::= \lambda x.v \mid x\, v_1 \ldots v_n$$

The terms of the second form $x\, v_1 \ldots v_n$ intuitively correspond to computations which are "stuck" because we do not know the function which is to be applied to the arguments: we only have a variable x here and will only be able to perform the reduction when this variable is substituted by an actual abstraction. Those are called *neutral values* and can be described by the grammar

$$n ::= x \mid n\, v$$

where v is a value. With this notation, values can be described by

$$v ::= \lambda x.v \mid n$$

We can thus describe values as the following datatype:

```
  type value =
    | VAbs of string * value
    | VNeu of neutral
  and neutral =
    | NVar of string
    | NApp of neutral * value
```

Now, remember that our idea is to use the evaluation of the language. In order to do so, the trick consists in describing λ-term $\lambda x.t$ not as a pair consisting of the variable x and the term t, but as the function

$$u \mapsto t[u/x]$$

which to a term u associates the term t with occurrences of x replaced by u: after all, the only thing we want to be able to perform with the λ-term $\lambda x.t$ is β-reduction! Instead of the above type, we thus actually describe values in this way by

```
type value =
  | VAbs of (value -> value)
  | VNeu of neutral
and neutral =
  | NVar of string
  | NApp of neutral * value
```

We can then implement a function which evaluates a term to a value as follows:

```
let rec eval env = function
  | Var x ->
    (try List.assoc x env with Not_found -> VNeu (NVar x))
  | Abs (x, t) -> VAbs (fun v -> eval ((x,v)::env) t)
  | App (t, u) -> vapp (eval env t) (eval env u)
and vapp v w =
  match v with
  | VAbs f -> f w
  | VNeu n -> VNeu (NApp (n, w))
```

The function `eval` takes as second argument the term to be evaluated (i.e. normalized) and, as first argument, an "environment" `env` which is a list of pairs (x, v) consisting of a variable x and a value v, such a pair indicating that the free variable x has to be replaced by the value v in the term during the evaluation (initially, this environment will typically be the empty list `[]`). We then evaluate terms as follows:

  – if the term is a variable x, we try to look up in the environment if there is a value for it, in which case we return it, and return the variable x otherwise,

  – if the term is an abstraction $\lambda x.t$, we return the value which is the function which to a value v associates the evaluation of t in the environment where x is bound to v,

  – if the term is an application $t\,u$, we evaluate t to a value $\hat{t}$ and u to a value $\hat{u}$; depending on the form of $\hat{t}$, we have two cases:

    – if $\hat{t} = f$ is a function, we simply apply it to $\hat{u}$,
    – if $\hat{t} = x\,v_1 \ldots v_n$, we return $x\,v_1 \ldots v_n\,\hat{u}$,

  this last part being taken care of by the auxiliary function `vapp` (which applies a value to another).

Finally, the environment is only really used during the evaluation and we define

```
let eval t = eval [] t
```

because from now on we will only use it in the empty environment.

You should note that an abstraction is not evaluated right away; instead, we construct a function which will evaluate it when an argument is given. In this sense, the function actually computes the weak normal form for the term. This function will not terminate if the β-reduction of the term does not. For instance, the evaluation of Ω will not terminate:

```
let _ = eval omega
```

Readback. We are now pleased because we have a short and efficient implementation of normalization, except for one point: we cannot easily print or serialize values because they contain functions. We now explain how we can convert a value back to a term: this procedure is called *readback*. We will need an infinite countable pool of fresh variables, so that we define the function

```
let fresh i = "x@" ^ string_of_int i
```

which generates the name of the i-th fresh variable, that we call here "x@i", supposing that initial terms will never contain variable names of this form (say that the user cannot use "@" in a variable name). The readback function can be implemented as follows:

```
let rec readback i v =
  (* Read back a neutral term. *)
  let rec neutral = function
    | NVar x -> Var x
    | NApp (n, v) -> App (neutral n, readback i v)
  in
  match v with
  | VAbs f ->
    let x = fresh i in
    Abs (x, readback (i+1) (f (VNeu (NVar x))))
  | VNeu n -> neutral n
```

It takes as argument an integer `i` (the index of the first fresh variable we have not used yet) and a value `v` and returns the term corresponding to the value:

- if the value is a function f, we return the term $\lambda x.t$ where $t = f(x)$ for some fresh variable x,

- otherwise it is of the form $x\,v_1 \ldots v_n$ and we return $x\,\overline{v}_1 \ldots \overline{v}_n$ where $\overline{v}_i$ is the term corresponding to the value v_i.

We can then define function which normalizes a term by evaluating it to a value and reading back the result:

```
let normalize t = readback 0 (eval t)
```

For instance, we can compute the normal form of the λ-term $(\lambda xy.x)y$, which is $\lambda z.y$, by

```
let _ =
  let t = App (Abs ("x", Abs ("y", Var "x")), Var "y") in
  normalize t
```

which gives the expected result

```
Abs ("x@0", Var "y")
```

Note that this reduction requires α-converting the abstraction on y, and this was correctly taken care of for us here.

Equivalence. Finally, we can test for β-equivalence of two λ-terms by comparing their normal forms (see section 4.2.4):

```
let eq t u = (normalize t) = (normalize u)
```

This is not as obvious as is seems: it also takes care of α-conversion! Namely, the readback function does not "randomly" generate fresh variables, but incrementing the counter i starting from 0 when progressing into the term. Because of this, it canonically renames the variables. For instance, one can check that the functions $\lambda x.x$ and $\lambda y.y$ are equal

```
let () =
  let id =  Abs ("x", Var "x") in
  let id' = Abs ("y", Var "y") in
  assert (eq id id')
```

Namely, both identity functions are going to be normalized into the term

```
Abs ("x@0", Var "x@0")
```

The above equality normalizes two terms in order to compare them. It is not as efficient as it could be in the case where the two terms are not equivalent: we might ensure that two terms are not equivalent without fully normalizing them. In order to understand why, first observe the following:

Lemma 3.5.2.1. Given a term t, the normal form of a term

   – $\lambda x.t$ is necessarily of the form $\lambda x.\hat{t}$,

   – $x\,t$ is necessarily of the form $x\,\hat{t}$

where $\hat{t}$ is the normal form of t.

Proof. This follows from the facts that the only possible way to β-reduce a term $\lambda x.t$ (resp. $x\,t$) is of the form $\lambda x.t \longrightarrow_\beta \lambda x.t'$ (resp. $x\,t \longrightarrow_\beta x\,t'$) by the rule (β_λ) (resp. (β_r)). $\qquad\square$

For this reason, we know that two terms of the form $\lambda x.t$ and $x\,u$ are never β-convertible, no matter what the terms t and u are. In such a situation, there is thus no need to fully normalize the two terms to compare them. More generally, a term $x\,t_1 \ldots t_n$ is never equivalent to an abstraction. Based on this observation, we can implement the test of β-equivalence as follows:

```
let eq t u =
  (* Equality of values *)
  let rec veq i v w =
    match v, w with
    | VAbs f, VAbs g ->
      let x = VNeu (NVar (fresh i)) in
      veq (i+1) (f x) (g x)
    | VNeu m, VNeu n -> neq i m n
    | _, _ -> false
  (* Equality of neutral terms *)
  and neq i m n =
    match m, n with
```

```
    | NVar x, NVar y -> x = y
    | NApp (m, v), NApp (n, w) -> neq i m n && veq i v w
    | _, _ -> false
  in
  veq 0 (eval t) (eval u)
```

Given two terms t and u, we reduce them to their *weak* normal form, i.e. we reduce them until we find abstractions:

– if they are of the form $\lambda x.t'$ and $x\,u_1\ldots u_n$ (or conversely), we know that they are not equivalent (even though we have not computed the normal form for t),

– if they are of the form $\lambda x.t'$ and $\lambda x.u'$ then we compare t' and u' (which requires evaluating them further)

– if they are of the form $x\,t_1\ldots t_m$ and $y\,u_1\ldots u_m$, where the t_i and u_i are weak normal forms, then they are equivalent if and only if $x = y$, $m = n$ and $t_i = u_i$ for every index i.

For instance, this procedure allows ensuring that $\lambda x.\Omega$ is not convertible to x:

```
let () =
  let t = Abs ("x", omega) in
  assert (not (eq t (Var "x")))
```

whereas the former equality procedure would loop when comparing the two terms because it tries to fully evaluate $\lambda x.\Omega$.

3.6 Nameless syntaxes

It might be difficult to believe at first, but a great source of bugs in software implementing compilers, proof-assistants, proving functional programs, and so on, comes from the incorrect handling of α-conversion. For instance, a naive implementation of substitution is:

$$
\begin{aligned}
x[u/x] &= u \\
y[u/x] &= y & \text{when } y \neq x \\
(t\,t')[u/x] &= (t[u/x])\,(t'[u/x]) \\
(\lambda y.t)[u/x] &= \lambda y.t[u/x]
\end{aligned}
$$

The last case is incorrect for two reasons. Firstly, we have to suppose that $x \neq y$, otherwise, the variables x inside t are not free, but rather bound by the abstraction, and should thus not be substituted: this case should be $(\lambda x.t)[u/x] = \lambda x.t$. Secondly, we also have to suppose $y \notin \mathrm{FV}(t)$: we are substituting x by u under the abstraction λy without taking in account the fact that y might get bound in y in this way. For instance, this implementation would lead to the following sequence of β-reductions

$$(\lambda y.yy)(\lambda fx.fx) \longrightarrow (\lambda fx.fx)(\lambda fx.fx) \longrightarrow \lambda x.(\lambda fx.fx)x \longrightarrow \lambda xx.xx$$

In the second reduction step, there is an erroneous capture of x: a correct normal form for the above term is $\lambda xy.xy$. There are multiple ways around

this, and we have already seen in section 3.5.2 that normalization by evaluation provides a satisfactory answer to this question. However, there are cases where this technique is not an option (e.g. the host language is not functional, or we want to perform more subtle manipulations than simply normalizing terms, or we want to formalize λ-calculus in a proof assistant). We present below some alternative syntaxes for λ-calculus which allow taking care of α-conversion in terms and implement β-reduction correctly and efficiently.

3.6.1 The Barendregt convention. A first idea in order to avoid incorrect captures of variables is to use the so-called *Barendregt convention* for naming the variables of λ-terms: all variables which are λ-abstracted should be pairwise distinct and distinct from all free variables.

Lemma 3.6.1.1. Every term is α-equivalent to one satisfying the Barendregt convention.

This convention sometimes simplifies things. For instance, the above naive implementation of β-reduction works on terms satisfying the convention. However, after one β-reduction step, the λ-term is not guaranteed to satisfy the Barendregt convention anymore (see the above example) and it is quite expensive to have to α-convert the whole term at each reduction step in order to enforce the convention.

3.6.2 De Bruijn indices. A more serious solution to this problem is given by de Bruijn indices. The idea is that, in a closed term, every variable is created by a specific abstraction in the term, so that instead of referring to a variable by its name, we can identify it by the abstraction which created it. Moreover, it turns out that there is a very convenient way to refer to an abstraction: the number of abstractions we have to step over when going up in the syntactic tree starting from the variable in order to reach the corresponding abstraction. This number is called the *de Bruijn index* of the variable. For instance, consider the λ-term

$$\lambda x.x(\lambda y.yx)$$

This lambda term can be graphically represented as a tree where a node labeled "λx" corresponds to an abstraction and a node "@" corresponds to an application:
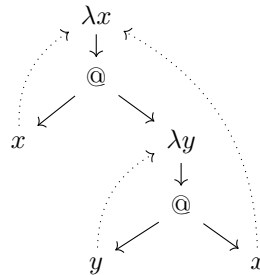
we have also figured in dotted arrows the links between a variable and the abstraction which created it. In the first variables x and y, the abstraction we are referring to is the one immediately above (we have to skip 0 λ's), whereas in the last occurrence of x, when going up starting from x in the syntactic tree, the corresponding abstraction is not the first one (which is λy) but the second

one (we have to skip 1 λ). The information in the λ-term can thus equivalently be represented by

$$\lambda x.0(\lambda y.01)$$

where each variable has been replaced by the number of λ's we have to skip when going up to reach the corresponding abstraction (note that a given variable, such as x above, can have different indices, depending on its position in the term). Now, the names of the variables do not really matter since we are working modulo α-conversion: we might as well drop them and simply write

$$\lambda.0(\lambda.01)$$

This is a very convenient notation because it does not mention variables anymore. What is not entirely clear yet is that we can implement β-reduction in this formalism. We will see that it is indeed possible, but quite subtle and difficult to get right.

Terms with de Bruijn indices. We thus consider a variant of the λ-calculus where terms are generated by the grammar

$$t, u ::= i \mid t\,u \mid \lambda.t$$

where $i \in \mathbb{N}$ is the de Bruijn index of a variable. Following the preceding remarks, a conversion function `of_term` from closed λ-terms into terms with de Bruijn indices is provided in figure 3.1. It takes an auxiliary argument `l` which is the list of variables already declared by abstractions: the de Bruijn index of a variable is then the index of the variable in this list.

The preceding function will raise the exception `Not_found` if the term contains free variables. It is however possible to adapt them to represent terms with free variables using de Bruijn indices. The idea is that we should represent a term t with n free variables $\mathrm{FV}(t) = \{x_0, \ldots, x_{n-1}\}$ as if we were computing the de Bruijn representation of t in the term $\lambda x_{n-1}.\ldots.\lambda x_0.t$, i.e. the free variables are implicitly abstracted. For instance

$$\lambda x.xx_0x_2 \qquad \text{is represented as} \qquad \lambda.013$$

In practice, it is also possible (and convenient) to "mix" the two conventions: have names for free variables and de Bruijn indices for bound variables. This is called the *locally nameless* representation of λ-terms [Cha12].

Reduction. Our goal is now to implement β-reduction in the de Bruijn representation of terms. The rule is, as usual,

$$(\lambda.t)u \longrightarrow_\beta t[u/0]$$

meaning that the variable 0 has to be replaced by u in t, where the substitution $t[u/0]$ remains to be defined. We actually need to define the substitution of any variable since, when going under an abstraction, the index of the variable to be substituted is increased by one. For instance, the β-reduction

$$\lambda x.(\lambda y.\lambda z.y)\,(\lambda t.t) \longrightarrow_\beta \lambda x.(\lambda z.y)[\lambda t.t/y] = \lambda x.\lambda z.y[\lambda t.t/y] = \lambda x.\lambda z.\lambda t.t$$

```
(** Traditional λ-terms. *)
type lambda =
  | LVar of string
  | LApp of lambda * lambda
  | LAbs of string * lambda

(** De Bruijn λ-terms. *)
type deBruijn =
  | Var of int
  | App of deBruijn * deBruijn
  | Abs of deBruijn

(** Index of an element in a list. *)
let rec index x = function
  | y::l -> if x = y then 0 else 1 + index x l
  | []   -> raise Not_found

(** De Bruijn representation of a closed term. *)
let of_term t =
  let rec aux l = function
    | LVar x      -> Var (index x l)
    | LApp (t, u) -> App (aux l t, aux l u)
    | LAbs (x, t) -> Abs (aux (x::l) t)
  in
  aux [] t
```

Figure 3.1: Converting λ-terms into de Bruijn representation.

i.e. graphically

$$
\begin{array}{ccc}
\lambda x & & \lambda x \\
\downarrow & & \downarrow \\
@ & & \lambda z \\
\swarrow \quad \searrow & & \downarrow \\
\lambda y \qquad \lambda t & \longrightarrow_\beta & \lambda t \\
\downarrow \qquad \downarrow & & \downarrow \\
\lambda z \qquad t & & t \\
\downarrow & & \\
y & &
\end{array}
$$
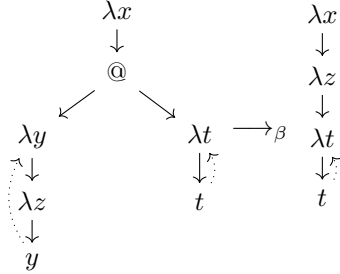
should correspond to the following steps

$$\lambda.(\lambda.\lambda.1)\,\lambda.0 \longrightarrow_\beta \lambda.(\lambda.1)[\lambda.0/0] = \lambda.\lambda.1[\lambda.0/1] = \lambda.\lambda.\lambda.0$$

We are thus tempted to define substitution by

$$
\begin{aligned}
i[u/i] &= u \\
j[u/i] &= j && \text{for } j \neq i \\
(t\,t')[u/i] &= (t[u/i])\,(t'[u/i]) \\
(\lambda.t)[u/i] &= \lambda.t[u/i+1]
\end{aligned}
$$

But it is incorrect because, in the last case, u might contain free variables, which refer to above abstractions, and have to be increased by 1 when going under the abstraction. For instance,

$$\lambda x.(\lambda y.\lambda z.y)\,x \longrightarrow_\beta \lambda x.(\lambda z.y)[x/y] = \lambda x.\lambda z.y[x/y] = \lambda x.\lambda z.x$$

i.e. graphically

$$
\begin{array}{ccc}
\lambda x & & \lambda x \\
\downarrow & & \downarrow \\
@ & & \lambda z \\
\swarrow \quad \searrow & & \downarrow \\
\lambda y \qquad x & \longrightarrow_\beta & x \\
\downarrow & & \\
\lambda z & & \\
\downarrow & & \\
y & &
\end{array}
$$
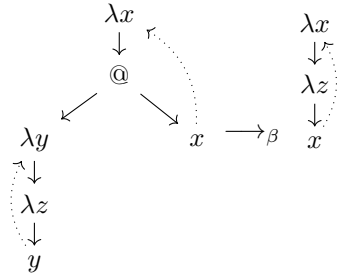
currently gives rise to the reduction

$$\lambda.(\lambda.\lambda.1)\,0 \longrightarrow_\beta \lambda.(\lambda.1)[0/0] = \lambda.\lambda.1[0/1] = \lambda.\lambda.0$$

whereas the correct reduction is

$$\lambda.(\lambda.\lambda.1)\,0 \longrightarrow_\beta \lambda.(\lambda.1)[0/0] = \lambda.\lambda.1[1/1] = \lambda.\lambda.1$$

The moral is that the last case of substitution should actually be

$$(\lambda.t)[u/i] = \lambda.t[u'/i+1] \tag{3.3}$$

where u' is the term obtained from u by increasing by 1 all free variables (and leaving other variables untouched), what we will write $u' = \uparrow_0 u$ in the following. The "corrected version" with (3.3) still contains a bug, which comes from the fact that β-reduction removes an abstraction, and therefore the indices of variables in t referring to the variables abstracted above the removed abstraction have to be decreased by 1. For instance,

$$\lambda x.(\lambda y.x)\,(\lambda t.t) \longrightarrow_\beta \lambda x.x[\lambda t.t/y] = \lambda x.x$$

i.e. graphically
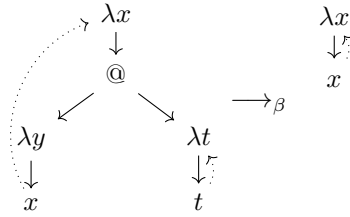
currently gives rise to the reduction

$$\lambda.(\lambda.1)\,(\lambda.0) \longrightarrow_\beta \lambda.1[\lambda.0/0] = \lambda.1$$

whereas the correct reduction is

$$\lambda.(\lambda.1)\,(\lambda.0) \longrightarrow_\beta \lambda.1[\lambda.0/0] = \lambda.0$$

This means that we should also correct the second case of substitution in order to decrease the index of variables which were free in the original substitution. And now we have it right.

In order to distinguish between bound and free variables in a term, it will be convenient to maintain an index l, called the *cutoff level*, such that the indices strictly below l correspond to bound variables and those above are free variables. We thus first define a function $\uparrow_l$ such that $\uparrow_l t$ is the term obtained from t by increasing by one all variables with index $i \geqslant l$, called the *lifting* of t at level l. By induction,

$$\uparrow_l i = \begin{cases} i & \text{if } i < l \\ i+1 & \text{if } i \geqslant l \end{cases}$$
$$\uparrow_l(t\,u) = (\uparrow_l t)\,(\uparrow_l u)$$
$$\uparrow_l(\lambda.t) = \lambda.(\uparrow_{l+1} t)$$

The right way to think about it is that $\uparrow_l t$ is the term obtained from t by adding a "new variable" of index l: the variables of index $i \geqslant l$ have to be increased by 1 in order to make room for the new variable. Similarly, we can define a function $\downarrow_l$ such that, for every term t which does not contain the variable l, $\downarrow_l t$ is the term obtained by removing the variable l (the *unlifting* of t): all variables of index $i > l$ have to be decreased by one. It turns out that we will only need it when t is a variable so that we define

$$\downarrow_l i = \begin{cases} i-1 & \text{if } i > l \\ i & \text{if } i < l \end{cases}$$

(it is not defined when $i = l$). With those at hand, we can finally correctly define substitution:

Definition 3.6.2.1 (Substitution). Given terms t and u and variable i, we define the substitution of i by u in t

$$t[u/i]$$

by induction by

$$
\begin{aligned}
i[u/i] &= u \\
j[u/i] &= \downarrow_i j && \text{for } j \neq i \\
(t\,t')[u/i] &= (t[u/i])\,(t'[u/i]) \\
(\lambda.t)[u/i] &= \lambda.t[\uparrow_0 u/i+1]
\end{aligned}
$$

As indicated above, β-reduction can then be implemented with the rule

$$(\lambda.t)u \longrightarrow_\beta t[u/0]$$

An implementation of call-by-value β-reduction (see section 3.5.1) on λ-terms in de Bruijn representation is given in figure 3.2.

3.6.3 Combinatory logic. *Combinatory logic*, which was introduced by Schönfinkel [Sch24], and further studied by Curry [Cur30, CF58], is another possible representation of λ-terms which does not need to use variable binding or α-conversion: in this syntax, there is simply no need for variables. Introductory references on the subject are [Bar84, chapter 7] and [Sel02].

Our starting point is the following question: is there a small number of "basic" λ-terms such that every λ-term can be obtained (up to β-equivalence) by applying the basic λ-terms one to the other. This would mean that all the abstractions we need in λ-terms can be generated from those contained in the basic λ-terms. It turns out that we only need three basic λ-terms, which encode some possible manipulations of variables:

- $\mathsf{I} = \lambda x.x$ corresponds to using a variable,

- $\mathsf{S} = \lambda xyz.(xz)(yz)$ corresponds to duplicating a variable,

- $\mathsf{K} = \lambda xy.x$ corresponds to erasing a variable.

It can be observed that the last abstracted variable of those terms is used, duplicated and erased respectively. As surprising as it seems as first, we can actually obtain any λ-term by application of those only. For instance, the term $\lambda xy.yy$ can be obtained as $\mathsf{K}\,((\mathsf{S}\,\mathsf{I})\,\mathsf{I})$: you can check that

$$\mathsf{K}\,((\mathsf{S}\,\mathsf{I})\,\mathsf{I}) \xrightarrow{\ *\ }_\beta \lambda xy.yy$$

Moreover, we can give the β-reduction rules directly for the basic terms: given any terms t, u, and v, we have

$$\mathsf{I}\,t \longrightarrow_\beta t \qquad\qquad \mathsf{S}\,t\,u\,v \longrightarrow_\beta (t\,v)(u\,v) \qquad\qquad \mathsf{K}\,t\,u \longrightarrow_\beta t$$

These are the only possible reductions for terms made of basic terms, and we have thus described β-reduction without using variables. This motivates the study, in the following of terms constructed from those, with the above rules as reduction.

```
(** Lambda terms. *)
type term =
  | Var of int
  | App of term * term
  | Abs of term

(** Lift a term at l. *)
let rec lift l = function
  | Var i      -> if i < l then Var i else Var (i + 1)
  | App (t, u) -> App (lift l t, lift l u)
  | Abs t      -> Abs (lift (l+1) t)

(** Unlift a variable i at l. *)
let unlift l i =
  assert (l <> i);
  if i < l then i else i-1

(** Substitute variable i for u in t. *)
let rec sub i u = function
  | Var j      -> if j = i then u else Var (unlift i j)
  | App (t, t') -> App (sub i u t, sub i u t')
  | Abs t      -> Abs (sub (i+1) (lift 0 u) t)

(** Call-by-value reduction. *)
let rec reduce = function
  | Var i      -> Var i
  | Abs t      -> Abs t
  | App (t, u) ->
    match reduce t with
    | Abs t' -> sub 0 (reduce u) t'
    | t      -> App (t, reduce u)
```

Figure 3.2: Normalization of λ-term using de Bruijn indices.

Definition. The terms of *combinatory logic* are generated by variables, application and the three above constants. Formally, they are generated by the grammar

$$T, U ::= x \mid T\,U \mid \mathsf{S} \mid \mathsf{K} \mid \mathsf{I}$$

where x is a variable, T and U are terms in combinatory logic and S, K and I are constants. The reduction rules are

$$\overline{\mathsf{S}\,T\,U\,V \longrightarrow (T\,V)\,(U\,V)} \qquad \overline{\mathsf{K}\,T\,U \longrightarrow T} \qquad \overline{\mathsf{I}\,T \longrightarrow T}$$

$$\frac{T \longrightarrow T'}{T\,U \longrightarrow T'\,U} \qquad\qquad \frac{U \longrightarrow U'}{T\,U \longrightarrow T\,U'}$$

We implicitly bracket application on the left, i.e. $T\,U\,V$ is read as $(T\,U)\,V$. As usual, we write $\overset{*}{\longrightarrow}$ for the reflexive and transitive closure of the relation $\longrightarrow$, and $\overset{*}{\longleftrightarrow}$ for it reflexive, symmetric and transitive closure. A *normal form* is a term which does not reduce. We write $\mathrm{FV}(T)$ for the set of variables of a term T. A *combinator* is a term without variables.

Implementation. In OCaml, the terms of combinatory logic can be described by the type

```
type term =
  | Var of var
  | App of term * term
  | S | K | I
```

The leftmost outermost reduction strategy (see section 3.5.1) can be shown to be normalizing: if a term admits a normal form then this strategy will reach it (we will see in theorem 3.6.3.3 that this normal form is necessarily unique). In OCaml, it can be implemented as follows:

```
let rec normalize t =
  match t with
  | Var _ | S | K | I -> t
  | App (t, v) ->
    match normalize t with
    | I -> v
    | App (K, t) -> normalize t
    | App (App (S, t), u) ->
      normalize (App (App (t, v), App (u, v)))
    | t -> App (t, normalize v)
```

An alternative, more elegant and efficient, implementation of this normalization procedure can be achieved by taking an additional argument env, which is a list of arguments the current term is applied to, which is sometimes called *Krivine's trick*: compared to the previous implementation, we avoid normalizing multiple times the same term.

```
let rec normalize t env =
  match t, env with
  | App (t, u), _ -> normalize t (u::env)
```

```
| I, t::env -> normalize t env
| K, t::u::env -> normalize t env
| S, t::u::v::env -> normalize t (v::(App(u,v))::env)
| t, env -> (* apply to normalized arguments *)
  List.fold_left (fun t u -> App (t, normalize u [])) t env
```

Example 3.6.3.1. Consider the combinator $\mathsf{S}\,\mathsf{K}\,\mathsf{K}$. It satisfies, for any term T,

$$\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,T \longrightarrow \mathsf{K}\,T\,(\mathsf{K}\,T) \longrightarrow T$$

Therefore the combinatory I is superfluous in our system, since it can be implemented as

$$\mathsf{I} = \mathsf{S}\,\mathsf{K}\,\mathsf{K}$$

which means that we could have restricted ourselves to the two combinators S and K only.

Example 3.6.3.2. The term $(\mathsf{S}\,\mathsf{I}\,\mathsf{I})\,(\mathsf{S}\,\mathsf{I}\,\mathsf{I})$ leads to an infinite sequence of reductions:

$$(\mathsf{S}\,\mathsf{I}\,\mathsf{I})\,(\mathsf{S}\,\mathsf{I}\,\mathsf{I}) \longrightarrow (\mathsf{S}\,\mathsf{I}\,\mathsf{I})\,(\mathsf{I}\,(\mathsf{S}\,\mathsf{I}\,\mathsf{I})) \overset{*}{\longrightarrow} (\mathsf{S}\,\mathsf{I}\,\mathsf{I})\,(\mathsf{S}\,\mathsf{I}\,\mathsf{I}) \longrightarrow \cdots$$

Theorem 3.6.3.3. The reduction relation $\overset{*}{\longrightarrow}$ is confluent.

Proof. This can be shown as in section 3.4, by introducing a notion of parallel reduction and showing that it has the diamond property. □

Abstraction. We can simulate abstractions in combinatory logic as follows. Given a variable x and a term T, we define a new term $\Lambda x.T$ by

$$\begin{aligned}
\Lambda x.x &= \mathsf{I} \\
\Lambda x.T &= \mathsf{K}\,T && \text{if } x \notin \mathrm{FV}(T), \\
\Lambda x.(T\,U) &= \mathsf{S}\,(\Lambda x.T)\,(\Lambda x.U) && \text{otherwise.}
\end{aligned}$$

Example 3.6.3.4. We have

$$\Lambda x.\Lambda y.x = \mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I}$$

Note that the term on the right is a normal form (in particular, it does not reduce to K).

Given terms T, U, we write $T[U/x]$ for the term T where the variable x has been replaced by U.

Lemma 3.6.3.5. For any terms T, U and variable x, we have

$$(\Lambda x.T)\,U \overset{*}{\longrightarrow} T[U/x]$$

Proof. By induction on T. □

Translation. We can now define translations between λ-terms and combinatory terms:

– we translate a λ-term t as a combinatory term $[\![t]\!]_{\mathrm{cl}}$,

– we translate a combinatory term T as a λ-term $[\![T]\!]_{\lambda}$.

These transformations are defined inductively as follows:

$$[\![x]\!]_{\mathrm{cl}} = x \qquad\qquad [\![x]\!]_{\lambda} = x$$
$$[\![t\,u]\!]_{\mathrm{cl}} = [\![t]\!]_{\mathrm{cl}}\,[\![u]\!]_{\mathrm{cl}} \qquad\qquad [\![T\,U]\!]_{\lambda} = [\![T]\!]_{\lambda}\,[\![U]\!]_{\lambda}$$
$$[\![\lambda x.t]\!]_{\mathrm{cl}} = \Lambda x.[\![t]\!]_{\mathrm{cl}} \qquad\qquad [\![\mathsf{S}]\!]_{\lambda} = \lambda xyz.(xz)(yz)$$
$$[\![\mathsf{K}]\!]_{\lambda} = \lambda xy.x$$
$$[\![\mathsf{I}]\!]_{\lambda} = \lambda x.x$$

Example 3.6.3.6. For instance, we have the following translations of λ-terms:

$$[\![\lambda xy.x]\!]_{\mathrm{cl}} = \mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I} \qquad\qquad [\![\lambda xy.yy]\!]_{\mathrm{cl}} = \mathsf{K}\,(\mathsf{S}\,\mathsf{I}\,\mathsf{I})$$

and of combinatory term:

$$[\![\mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I}]\!]_{\lambda} = (\lambda xyz.(xz)(yz))((\lambda xy.x)(\lambda xy.x))(\lambda x.x)$$

Lemma 3.6.3.7. For any terms T, U, if $T \overset{*}{\longrightarrow} U$ then $[\![T]\!]_{\lambda} \overset{*}{\longrightarrow}_{\beta} [\![U]\!]_{\lambda}$.

Proof. By induction on T. ☐

The reduction of combinatory terms can be simulated in λ-calculus:

Lemma 3.6.3.8. For any term T, $[\![\Lambda x.T]\!]_{\lambda} \overset{*}{\longrightarrow}_{\beta} \lambda x.[\![T]\!]_{\lambda}$.

Proof. By induction on T. ☐

Translating a λ-term back and forth has no effect up to β-equivalence:

Lemma 3.6.3.9. For any λ-term t, $[\![[\![t]\!]_{\mathrm{cl}}]\!]_{\lambda} \overset{*}{\longrightarrow}_{\beta} t$.

Proof. By induction on t, using previous lemma in the case of abstraction. ☐

The previous theorem, together with lemma 3.6.3.7, can be seen as the fact that combinatory logic embeds into λ-calculus (modulo β-reduction). It also implies that the basic combinators S, K and I can be thought of as a "basis" from which all the λ-terms can be generated:

Corollary 3.6.3.10. Every closed λ-term is β-equivalent to one obtained from S, K and I by application.

This correspondence between λ-calculus and combinatory logic unfortunately has a number of minor defects. First, it is not true that, for λ-terms t and u,

$$t \overset{*}{\longrightarrow}_{\beta} u \qquad \text{implies} \qquad [\![t]\!]_{\mathrm{cl}} \overset{*}{\longrightarrow} [\![u]\!]_{\mathrm{cl}}$$

For instance, we have

$$[\![\lambda x.(\lambda y.y)\,x]\!]_{\mathrm{cl}} = \mathsf{S}\,(\mathsf{K}\,\mathsf{I})\,\mathsf{I} \qquad\qquad [\![\lambda x.x]\!]_{\mathrm{cl}} = \mathsf{I} \qquad\qquad (3.4)$$

where both combinatory terms are normal forms. If we try to go through the induction, the problem comes from the fact that β-reduction satisfies the rule on the left below, often called (ξ), whereas the corresponding principle on the right is not valid in combinatory logic:

$$\frac{t \longrightarrow_{\beta} t'}{\lambda x.t \longrightarrow_{\beta} \lambda x.t'}\ (\xi) \qquad\qquad \frac{T \longrightarrow T'}{\Lambda x.T \longrightarrow \Lambda x.T'}$$

as the above example illustrates. Intuitively, this is due to the fact that we have not yet provided enough arguments to the terms. Namely, if we apply both terms of (3.4) to an arbitrary term T, we obtain the same result:

$$\mathsf{S}\,(\mathsf{K}\,\mathsf{I})\,\mathsf{I}\,T \longrightarrow \mathsf{K}\,\mathsf{I}\,T\,(\mathsf{I}\,T) \longrightarrow \mathsf{I}\,(\mathsf{I}\,T) \longrightarrow \mathsf{I}\,T \longrightarrow T \qquad \text{and} \qquad \mathsf{I}\,T \longrightarrow T$$

In general, it can be shown that

$$t \xrightarrow{*}_\beta u \qquad \text{implies} \qquad [\![t]\!]_{\mathrm{cl}}\,T_1 \dots T_n \xrightarrow{*} [\![u]\!]_{\mathrm{cl}}\,T_1 \dots T_n$$

for every terms T_i, provided that n is a large enough natural number depending on t and u. It is also not true that the translation of a combinatory term in normal form is a normal λ-term:

$$[\![\mathsf{K}\,x]\!]_{\mathrm{cl}} = (\lambda xy.x)\,x \longrightarrow_\beta \lambda y.\,x$$

Again, the term $\mathsf{K}\,x$ is intuitively a normal form only because it is not applied to enough arguments. Finally, given a term T, the terms $[\![[\![T]\!]_\lambda]\!]_{\mathrm{cl}}$ and T are not convertible in general. For instance

$$[\![[\![\mathsf{K}]\!]_\lambda]\!]_{\mathrm{cl}} = [\![\lambda xy.x]\!]_{\mathrm{cl}} = \mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I} \neq \mathsf{K}$$

Both terms are normal forms and if they were convertible, they would reduce to a common term by theorem 3.6.3.3. This is again due to the lack of arguments: for every term T, we have

$$\mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I}\,T \xrightarrow{*} \mathsf{K}\,T$$

Two combinatory terms T and T' are *extensionally equivalent*, when for every term U, we have $T\,U \xleftrightarrow{*} T'\,U$. It can be shown that combinatory terms modulo reduction and extensional equivalence are in bijection with λ-terms modulo β and η, via the translations we have defined.

Iota. We have seen in example 3.6.3.1 that the combinatory I is superfluous, so that the two combinators S and K are sufficient. Can we remove another combinator? With S and K we cannot. We can however come up with one combinator which subsumes both S and K: if we define the λ-term

$$\iota = \lambda x.x\,\mathsf{S}\,\mathsf{K}$$

we have

$$\mathsf{I} = \iota\,\iota \qquad\qquad \mathsf{K} = \iota\,(\iota\,(\iota\,\iota)) \qquad\qquad \mathsf{S} = \iota\,(\iota\,(\iota\,(\iota\,\iota)))$$

We can therefore base combinatory logic on the only combinator ι, the reduction rule being

$$\iota\,T \longrightarrow T\,\mathsf{S}\,\mathsf{K} = T\,(\iota\,(\iota\,(\iota\,(\iota\,\iota))))\,(\iota\,(\iota\,(\iota\,\iota)))$$

In the sense described above, any λ-term can thus be encoded as a combinator based on ι, i.e. as a term generated by the grammar

$$t, u ::= \iota \mid t\,u$$

Any λ-term t can thus be encoded as a binary word $[t]$ defined by

$$[\iota] = 1 \qquad\qquad\qquad [t\,u] = 0[t][u]$$

so that $\iota\,(\iota\,(\iota\,\iota))$ is encoded as 0101011.

Simply typed λ-calculus

If λ-calculus introduced in chapter 3 can be seen as the functional core of a programming language, the simply typed λ-calculus studied in this chapter is the core of a *typed* programming language. It will allow us to give a formal meaning to the title of the book: we will see that a type can be seen as a formula and a typable λ-term corresponds precisely to a proof of its type. This is the so-called *Curry-Howard correspondence* which is at the heart of this course. From a historical point of view, this calculus was introduced by Church in the 40s [Chu40], in order to provide a foundation of logics and mathematics. Good further reading material on the subject include [Pie02, SU06].

We introduce types for λ-calculus in section 4.1, show that typable terms are terminating in section 4.2, extend typing to types constructors other than arrows in section 4.3, discuss the variant where abstracted variables are not typed in section 4.4, discuss the relationship between Hilbert calculus and combinators in section 4.5, and finally present extensions to classical logic in section 4.6.

4.1 Typing

4.1.1 Types. A *simple type* is an expression made of variables and arrows. Those are generated by the grammar

$$A, B ::= X \mid A \to B$$

A simple type is thus either

- a type variable X,

- an arrow type $A \to B$ read as the type of functions from A to B (which are themselves simple types).

By convention arrows are implicitly bracketed on the right: $A \to B \to C$ is read as $A \to (B \to C)$.

4.1.2 Contexts. A *context*

$$\Gamma = x_1 : A_1, \ldots, x_n : A_n$$

is a list of pairs consisting of a variable x_i (in the sense of λ-calculus, see section 3.1.1) and a type A_i. A context is thus either the empty context or of the form $\Gamma, x : A$ for some context Γ, which is useful to reason by induction on contexts. The *domain* $\mathrm{dom}(\Gamma)$ of the context Γ is the set of variables occurring in it:

$$\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$$

Given a variable $x \in \mathrm{dom}(\Gamma)$, we sometimes write $\Gamma(x)$ for the type associated with it. Here, we do not require that in a context all the variables x_i are

distinct: to be precise $\Gamma(x)$ is the rightmost pair $x : A$ occurring in Γ, which can be defined by induction by

$$(\Gamma, x : A)(x) = A \qquad\qquad (\Gamma, y : A)(x) = \Gamma(x)$$

for $y \neq x$.

4.1.3 λ-terms. We are going to consider a small variation of λ-terms: we suppose that all λ-abstractions specify the type of the abstracted variable. The syntax for terms is thus

$$t, u ::= x \mid t\,u \mid \lambda x^A.t$$

where x is a variable, t and u are terms, and A is a type. An abstraction $\lambda x^A.t$ should be read as a function taking an argument x *of type A* and returning t.

Church vs Curry style for λ-terms. The above convention, where abstractions are typed, is called *Church style* λ-terms. We will see that adopting it greatly simplifies the questions one is usually interested in for those terms (such as type checking, see section 4.1.6), at the cost of requiring small annotations from the user (the type of the abstractions).

A variant of the theory where abstractions are not typed can also be developed and is called *Curry style*, see section 4.4. This is for instance the convention used in OCaml: one would typically write

```
let f = fun x -> x
```

although the Church style is also supported, i.e. we can also write

```
let f = fun (x:int) -> x
```

4.1.4 Typing. A *sequent* is a triple written as

$$\Gamma \vdash t : A \tag{4.1}$$

consisting of context Γ, a λ-term t and a type A. A term t *has type A* in a context Γ when the sequent (4.1) is derivable using the three rules of figure 4.1 where, in the rule (ax), we suppose $x \in \mathrm{dom}(\Gamma)$ satisfied as a side condition. Those rules can be read as follows:

– (ax): in an environment where x is of type A, we know that x is of type A,

– ($\to_{\mathrm{I}}$): if, supposing x is of type A, t is of type B, then the function $\lambda x.t$ which to x associates t is of type $A \to B$,

– ($\to_{\mathrm{E}}$): given a function t of type $A \to B$ and an argument u of type A, the result of the application $t\,u$ is of type B.

We simply say that the term t has type A if it is so in the empty context. A derivation in this system is sometimes called a *typing derivation*. A term t is *typable* when it has some type A in some context Γ.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \ (\text{ax})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \to B} \ (\to_{\text{I}})$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B} \ (\to_{\text{E}})$$

Figure 4.1: Typing rules of simply-typed λ-calculus

Example 4.1.4.1. The term

$$\lambda f^{A \to A}.\lambda x^A.f(fx)$$

has type

$$(A \to A) \to A \to A$$

Namely, we have the typing derivation

$$\frac{\dfrac{}{\Gamma \vdash f : A \to A} \ (\text{ax}) \qquad \dfrac{\dfrac{}{\Gamma \vdash f : A \to A} \ (\text{ax}) \qquad \dfrac{}{\Gamma \vdash x : A} \ (\text{ax})}{\Gamma \vdash fx : A} \ (\to_{\text{E}})}{\dfrac{\dfrac{f : A \to A, x : A \vdash f(fx) : A}{f : A \to A \vdash \lambda x^A.f(fx) : A \to A} \ (\to_{\text{I}})}{\vdash \lambda f^{A \to A}.\lambda x^A.f(fx) : (A \to A) \to A \to A} \ (\to_{\text{I}})} \ (\to_{\text{E}})$$

with

$$\Gamma = f : A \to A, x : A$$

Remark 4.1.4.2. Although this will mostly remain implicit in the following, we consider sequents up to α-conversion: this means that, in a sequent $\Gamma \vdash t : A$, we can change a variable x into y both in Γ and in t at the same time, provided that $y \notin \text{dom}(\Gamma)$. Because of this, we can always assume that all the variables are distinct in the contexts we consider. This assumption is sometimes useful to reason about proofs, e.g. with this convention, the axiom rule is equivalent to

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \ (\text{ax})$$

We do however feel bad about systematically assuming this because, in practice, implementations of logical or typing systems do not maintain this invariant.

4.1.5 Basic properties of the typing system. We state here some basic properties of the typing system, which will be used later on. First, the following variant of the structural rules (see section 2.2.10) hold.

Lemma 4.1.5.1 (Weakening rule)*.* The *weakening rule* is admissible

$$\frac{\Gamma, \Gamma' \vdash t : B}{\Gamma, x : A, \Gamma' \vdash t : B} \ (\text{wk})$$

provided that $x \notin \text{dom}(\Gamma)$.

Proof. By induction on the derivation of $\Gamma, \Gamma' \vdash t : B$. The case of axiom rule uses the fact that we can suppose that $x \notin \text{dom}(\Gamma)$, since we are considering sequents up to α-conversion, see remark 4.1.4.2. □

Lemma 4.1.5.2 (Exchange rule)*.* The *exchange rule* is admissible

$$\frac{\Gamma, x : A, y : B, \Gamma' \vdash t : C}{\Gamma, y : B, x : A, \Gamma' \vdash t : C} \ (\text{xch})$$

provided that $x \neq y$.

Proof. By induction on the derivation of the premise. □

Lemma 4.1.5.3 (Contraction rule)*.* The *contraction rule* is admissible:

$$\frac{\Gamma, x : A, y : A, \Gamma' \vdash t : B}{\Gamma, x : A, \Gamma' \vdash t[x/y] : B} \ (\text{contr})$$

Proof. By induction on the derivation of the premise. □

All the free variables of a typable term are bound in the context:

Lemma 4.1.5.4*.* Given a sequent $\Gamma \vdash t : A$ which is derivable, we have $\text{FV}(t) \subseteq \text{dom}(\Gamma)$.

Proof. By induction on the derivation of the sequent. □

In particular, a term t typable in the empty context is necessarily closed, i.e. $\text{FV}(t) = \emptyset$. Conversely, a variable which does not occur in the term can be removed:

Lemma 4.1.5.5*.* Given a derivable sequent $\Gamma, x : A, \Gamma' \vdash t : A$ with $x \notin \text{FV}(t)$, the sequent $\Gamma, \Gamma' \vdash t : A$ is also derivable.

Proof. By induction on the derivation of the sequent. □

4.1.6 Type checking, type inference and typability. The three most important algorithmic questions when considering a typing system are the following ones.

– The *type checking* problem consists, given a context Γ, a term t and a type A, in deciding whether t has type A in context Γ.

– The *type inference* problem consists, given a context Γ and a term t which is typable in the context Γ, in finding a type A such that t has type A in context Γ.

– The *typability* problem consists, given a context Γ and a term t, in deciding whether t admits a type in this context.

In simply-typed λ-calculus all those three problems are very easy: they can be answered in linear time over the size of the term t (neglecting the size of Γ):

Theorem 4.1.6.1 (Uniqueness of typing). *Given a context Γ and a term t there is at most one type A such that t has type A in the context Γ and at most one derivation of $\Gamma \vdash t : A$.*

Proof. By induction on the term t. We have the following cases depending on its shape:

– if the term is of the form x then it is typable iff $x \in \mathrm{dom}(\Gamma)$ and in this case the typing derivation is

$$\frac{}{\Gamma \vdash x : A} \; (\mathrm{ax})$$

with $A = \Gamma(x)$,

– if the term is of the form $t\,u$ then it is typable iff both t and u are typable in Γ, with respective types of the form $A \to B$ and A, and in this case the typing derivation is

$$\frac{\begin{array}{c} \vdots \\ \overline{\Gamma \vdash t : A \to B} \end{array} \qquad \begin{array}{c} \vdots \\ \overline{\Gamma \vdash u : A} \end{array}}{\Gamma \vdash t\,u : B} \; (\to_{\mathrm{E}})$$

– if the term is of the form $\lambda x^A.t$ then it is typable iff t is typable in context $\Gamma, x : A$ with some type B, and in this case the typing derivation is

$$\frac{\begin{array}{c} \vdots \\ \overline{\Gamma, x : A \vdash B} \end{array}}{\Gamma \vdash \lambda x^A.t : A \to B} \; (\to_{\mathrm{I}})$$

This concludes the proof. $\qquad\square$

The above theorem allows one to speak of "the" type and "the" typing derivation of a typable term. Moreover, its proof is constructive, in the sense that it allows to explicitly construct the type of a term when it exists (i.e. perform type inference) and determine that the type admits no type otherwise (i.e. perform typability), by induction on the type of the term. Since a term admits a unique type, the type checking problem can be reduced to type inference: in a given context, a term t admits a type A if and only if the type inferred for t is A.

Implementation. An implementation is provided in figure 4.2.

– The function `infer` infers the type of a given term in a given context `env`, which is a list of pairs consisting of a variable and a type, encoding the typing context Γ (in reverse order). Depending on whether the term is a variable, an abstraction or an application, the function will recursively look for proofs, using the rules (ax), ($\to_{\mathrm{I}}$) and ($\to_{\mathrm{E}}$) respectively. The function raises the exception `Not_found` when no such type exists.

– The function `check` performs type checking: given an environment `env`, a term `t` and a type `a`, it returns `()` if the term admits the given type and raises `Not_found` otherwise. The implementation of this function corresponds to the proof of theorem 4.1.6.1.

– The function `typable` determines whether a terms admits a type or not in a given environment.

4.1.7 The Curry-Howard correspondence. The presentation and naming of the rules of section 4.1.4 is intended to make it clear the relation with logic: if we erase the term annotations and replace $\to$ by $\Rightarrow$, we obtain precisely the rules of the implicational fragment of intuitionistic logic, see section 2.2.6. This parallel between the typing rules (on the left) and the rules in natural deduction (on the right) is shown in the table below:

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}\ (\text{ax}) \qquad\qquad \frac{}{\Gamma, A, \Gamma' \vdash A}\ (\text{ax})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \to B}\ (\to_{\text{I}}) \qquad\qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ (\Rightarrow_{\text{I}})$$

$$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}\ (\to_{\text{E}}) \qquad\qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\Rightarrow_{\text{E}})$$

If we start from a typing derivation, we obtain a derivation in NJ by erasing the terms, i.e. replacing a rule on the left column above by the corresponding rule on the right column: this process is called here the *term erasing procedure*. Abstractions thus correspond to introduction rules of $\Rightarrow$, applications to elimination rules of $\Rightarrow$, and variables to axiom rules. In fact, the relationship between typable terms and proofs in NJ is very tight: this is known as the *Curry-Howard correspondence*, also called *proofs-as-programs correspondence* or *propositions-as-types correspondence*. It was first explicitly stated by Howard in notes which circulated starting from 1969 and were ultimately published a decade later [How80]. The name of Curry is due to his closely related discovery of the correspondence between Hilbert calculus and combinatory logic [CF58] in the late 50s, detailed in section 4.5. We recall that natural deduction [Gen35] and simply typed λ-calculus [Chu40] were introduced in 1935 and 1940: this correspondence might look like an obvious fact once the concepts are properly elaborated and the right notations set up, but it took 30 years to get there.

Theorem 4.1.7.1 (Curry-Howard correspondence). Given a context Γ and a type A, the term erasing procedure induces a one-to-one correspondence between

(i) λ-terms of type A in the context Γ, and

(ii) proofs in the implicational fragment of NJ of $\Gamma \vdash A$.

Proof. Suppose given a proof π of a sequent. We construct a term t having this proof as typing derivation by induction on the derivation:

```
type var = string

(** Types. *)
type ty =
  | TVar of string
  | Arr  of ty * ty

(** Terms. *)
type term =
  | Var of var
  | App of term * term
  | Abs of var  * ty * term

exception Type_error

(** Type inference. *)
let rec infer env = function
  | Var x ->
    (try List.assoc x env with Not_found -> raise Type_error)
  | Abs (x, a, t) ->
    Arr (a, infer ((x,a)::env) t)
  | App (t, u) ->
    match infer env t with
    | Arr (a, b) -> check env u a; b
    | _ -> raise Type_error

(** Type checking. *)
and check env t a =
  if infer env t <> a then raise Type_error

(** Typability. *)
let typable env t =
  try  let _ = infer env t in true
  with Type_error -> false
```

Figure 4.2: Type checking, type inference and typability.

– if the proof is of the form

$$\frac{}{\Gamma, A, \Gamma' \vdash A}(\text{ax})$$

then necessarily the corresponding typing derivation is

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}(\text{ax})$$

– if the proof is of the form

$$\frac{\begin{array}{c}\pi \\ \overline{\Gamma \vdash A \Rightarrow B}\end{array} \quad \begin{array}{c}\pi' \\ \overline{\Gamma \vdash A}\end{array}}{\Gamma \vdash B}(\Rightarrow_{\text{E}})$$

then by induction hypothesis we have a terms t and u with typing derivations

$$\begin{array}{c}\vdots \\ \overline{\Gamma \vdash t : A \to B}\end{array} \qquad \begin{array}{c}\vdots \\ \overline{\Gamma \vdash u : A}\end{array}$$

and necessarily the typing derivation is

$$\frac{\begin{array}{c}\vdots \\ \overline{\Gamma \vdash t : A \to B}\end{array} \quad \begin{array}{c}\vdots \\ \overline{\Gamma \vdash u : A}\end{array}}{\Gamma \vdash t\,u : B}(\to_{\text{I}})$$

– if the proof is of the form

$$\frac{\begin{array}{c}\pi \\ \overline{\Gamma, A \vdash B}\end{array}}{\Gamma \vdash A \Rightarrow B}(\Rightarrow_{\text{I}})$$

then by induction hypothesis we have a typing derivation

$$\begin{array}{c}\vdots \\ \overline{\Gamma, x : A \vdash t : B}\end{array}$$

and necessarily the typing derivation is of the form

$$\frac{\begin{array}{c}\vdots \\ \overline{\Gamma, x : A \vdash t : B}\end{array}}{\Gamma \vdash \lambda x^A.t : A \to B}(\to_{\text{I}})$$

Conversely, given a term of type A in the context Γ, theorem 4.1.6.1 ensures that there is at most one type derivation for it, and erasing it provides a proof of $\Gamma \vdash A$. Finally, it is easily shown that both translations establish a bijective correspondence. $\qquad\square$

In the light of the previous theorem, typable λ-terms can be thought of as *witnesses* for proofs.

Remark 4.1.7.2 (Contexts as sets). The two λ-terms $\lambda x^A.\lambda y^A.x$ and $\lambda x^A.\lambda y^A.y$ both have the type $A \to A \to A$:

$$\frac{\dfrac{\overline{x : A, y : A \vdash x : A}\ \text{(ax)}}{\dfrac{x : A \vdash \lambda y^A.x : A \to A}{\vdash \lambda x^A.\lambda y^A.x : A \to A \to A}\ (\to_{\text{I}})}\ (\to_{\text{I}})} \qquad \frac{\dfrac{\overline{x : A, y : A \vdash y : A}\ \text{(ax)}}{\dfrac{x : A \vdash \lambda y^A.y : A \to A}{\vdash \lambda x^A.\lambda y^A.y : A \to A \to A}\ (\to_{\text{I}})}\ (\to_{\text{I}})}$$

and they are clearly different (they respectively correspond to the first and the second projection). This sheds a new light on our remark of section 2.2.10, stating that contexts should be lists and not sets in proof systems. If we handled them as sets, we would not be able to distinguish them since both would correspond, via the "Curry-Howard correspondence", to the proof

$$\frac{\dfrac{\overline{A \vdash A}\ \text{(ax)}}{\dfrac{A \vdash A \Rightarrow A}{\vdash A \Rightarrow A \Rightarrow A}\ (\Rightarrow_{\text{I}})}\ (\Rightarrow_{\text{I}})}$$

In other words, it is important, in axiom rules, to know exactly which hypothesis we are using in the context when there are two of the same type.

Remark 4.1.7.3 (Equivalence vs isomorphism). In the same vein as previous remark, there is a difference between equivalence and isomorphism in type theory. For instance, we have an equivalence

$$(A \Rightarrow A \Rightarrow B) \Leftrightarrow (A \Rightarrow B)$$

but the types $A \Rightarrow A \Rightarrow B$ and $A \Rightarrow B$ are not isomorphic. The equivalence amounts to having terms corresponding to both implications of the equivalence:

$$t : (A \to A \to B) \to (A \to B) \qquad u : (A \to B) \to (A \to A \to B)$$

Here, we can take

$$t = \lambda f^{A \to A \to B}.\lambda x^A.f\, x\, x \qquad\qquad u = \lambda f^{A \to B}.\lambda x^A.\lambda y^A.f\, x$$

Such a pair of terms is an *isomorphism* when both composites are ($\beta\eta$-equivalent to) the identity:

$$\lambda f^{A \to B}.t\,(u\,f) =\!=_{\beta\eta} \lambda f^{A \to B}.f$$
$$\lambda f^{A \to A \to B}.u\,(t\,f) =\!=_{\beta\eta} \lambda f^{A \to A \to B}.f$$

In the above example, the first equality does hold, but not the second since

$$\lambda f^{A \to A \to B}.u\,(t\,f) = \lambda f^{A \to A \to B}.\lambda x^A.f\, x\, x \neq_{\beta\eta} \lambda f^{A \to A \to B}.f$$

4.1.8 Subject reduction. An important property, relating typing and β-reduction in the λ-calculus is the *subject reduction property*, already encountered in theorem 1.4.3.2: typing does not change during evaluation, by which we mean here β-reduction. We first need an auxiliary lemma:

Lemma 4.1.8.1 (Substitution lemma). Suppose that we have a typing derivation of

$$\Gamma, x : A, \Gamma' \vdash t : B \qquad \text{and} \qquad \Gamma, \Gamma' \vdash u : A$$

then we have a typing derivation of $\Gamma, \Gamma' \vdash t[u/x] : B$. In other words, the rule

$$\frac{\Gamma, x : A, \Gamma' \vdash t : B \qquad \Gamma, \Gamma' \vdash u : A}{\Gamma, \Gamma' \vdash t[u/x] : B}$$

is admissible.

Proof. By induction on the typing derivation of $\Gamma, x : A, \Gamma' \vdash t : B$.

– If it is of the form

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \ (\text{ax})$$

then we conclude with the derivation of $\Gamma, \Gamma' \vdash u : A$ in the hypothesis.

– If it is of the form

$$\frac{}{\Gamma, x : A, \Gamma' \vdash y : B} \ (\text{ax})$$

where $x \neq y$ and $y : B$ occurs in Γ or Γ', then we conclude with

$$\frac{}{\Gamma, \Gamma' \vdash y : B} \ (\text{ax})$$

– If it is of the form

$$\frac{\dfrac{\pi_1}{\Gamma, x : A, \Gamma' \vdash t : B \Rightarrow C} \qquad \dfrac{\pi_2}{\Gamma, x : A, \Gamma' \vdash t' : B}}{\Gamma, x : A, \Gamma' \vdash t\,t' : C} \ (\Rightarrow_{\text{E}})$$

then we conclude with

$$\frac{\dfrac{\pi_1'}{\Gamma, \Gamma' \vdash t[u/x] : B \Rightarrow C} \qquad \dfrac{\pi_2'}{\Gamma, x : A, \Gamma' \vdash t'[u/x] : B}}{\Gamma, \Gamma' \vdash (t[u/x])\,(t'[u/x]) : C} \ (\Rightarrow_{\text{E}})$$

where π_1' and π_2' are respectively obtained from π_1 and π_2 by induction hypothesis.

– If it is of the form

$$\frac{\dfrac{\pi}{\Gamma, x : A, \Gamma', y : B \vdash t : C}}{\Gamma, x : A, \Gamma' \vdash \lambda y.t : B \Rightarrow C} \ (\Rightarrow_{\text{I}})$$

then we conclude with

$$\frac{\dfrac{\pi'}{\Gamma, \Gamma', y : B \vdash t[u/x] : C}}{\Gamma, \Gamma' \vdash \lambda y.t[u/x] : B \Rightarrow C} \ (\Rightarrow_{\text{I}})$$

where π' is obtained from

$$\dfrac{\pi}{\Gamma, x : A, \Gamma', y : B \vdash t : C} \qquad \text{and} \qquad \dfrac{\dfrac{\vdots}{\Gamma, \Gamma' \vdash u : A}}{\Gamma, x : A, \Gamma' \vdash u : A} \text{ (wk)}$$

by induction hypothesis. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

Remark 4.1.8.2. Note that, through the Curry-Howard correspondence, the substitution lemma precisely corresponds to the "proof substitution" of proposition 2.3.2.1: the term erasure of the rule of lemma 4.1.8.1 is the cut rule

$$\dfrac{\Gamma, A, \Gamma' \vdash B \qquad \Gamma, \Gamma' \vdash A}{\Gamma, \Gamma' \vdash B} \text{ (cut)}$$

It should not be a surprise: under the Curry-Howard correspondence, substituting proofs corresponds to substituting terms.

Theorem 4.1.8.3 (Subject reduction). Suppose given a term t of type A in a context Γ. If t β-reduces to t' then t' also has type A in the context Γ.

Proof. By induction on the derivation of $t \longrightarrow_\beta t'$ (see section 3.2.1).

– If the derivation ends with (β_s), it is of the form

$$(\lambda x.t)u \longrightarrow_\beta t[u/x]$$

and the typing derivation of the term on the left is of the form

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x : A \vdash t : B}}{\Gamma \vdash \lambda x.t : A \to B} (\to_I) \qquad \dfrac{\vdots}{\Gamma \vdash u : B}}{\Gamma \vdash (\lambda x.t)\, u : B} (\to_E)$$

We conclude by lemma 4.1.8.1 which ensures the existence of a derivation of the form

$$\dfrac{\vdots}{\Gamma \vdash t[u/x] : B}$$

– If the derivation ends with (β_l), it is of the form

$$t\, u \longrightarrow_\beta t'\, u$$

with $t \longrightarrow_\beta t'$, and the typing derivation of the term on the left is of the form

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash t : A \to B} \qquad \dfrac{\pi_2}{\Gamma, x : A \vdash u : B}}{\Gamma \vdash t\, u : B} (\to_E)$$

We conclude with the derivation

$$\dfrac{\dfrac{\pi_1'}{\Gamma \vdash t' : A \to B} \qquad \dfrac{\pi_2}{\Gamma, x : A \vdash u : B}}{\Gamma \vdash t'\, u : B} (\to_E)$$

where π_1' is obtained by induction hypothesis.

– The cases of (β_r) and (β_λ) are similar to the previous one. $\qquad\square$

Example 4.1.8.4. We have the typing derivation

$$
\cfrac{
  \cfrac{
    \cfrac{}{x : A, y : A \vdash y : A} \ (\text{ax})
  }{
    x : A \vdash \lambda y^A.y : A \to A
  } \ (\to_\text{I})
  \qquad
  \cfrac{}{x : A \vdash x : A} \ (\text{ax})
}{
  \cfrac{
    x : A \vdash (\lambda y^A.y)x : A
  }{
    \vdash \lambda x^A.(\lambda y^A.y)x : A \to A
  } \ (\to_\text{I})
} \ (\to_\text{E})
$$

and the reduction

$$\lambda x^A.(\lambda y^A.y)x \longrightarrow_\beta \lambda x^A.x$$

It can be checked that the reduced term does admit the same type $A \to A$:

$$
\cfrac{
  \cfrac{}{x : A \vdash x : A} \ (\text{ax})
}{
  \vdash \lambda x^A.x : A \to A
} \ (\to_\text{I})
$$

The proof of the above theorem deserves some attention. It should be observed that, by erasing the terms, the β-reduction of a typable term described in the above proof corresponds precisely to the procedure we used in section 2.3.3 in order to eliminate a cut in the corresponding proof:

$$
\cfrac{
  \cfrac{
    \cfrac{\vdots}{\Gamma, A \vdash B}
  }{
    \Gamma \vdash A \Rightarrow B
  } \ (\Rightarrow_\text{I})
  \qquad
  \cfrac{\vdots}{\Gamma \vdash A}
}{
  \Gamma \vdash B
} \ (\Rightarrow_\text{E})
\qquad \rightsquigarrow \qquad
\cfrac{\vdots}{\Gamma \vdash B}
$$

Thus,

Theorem 4.1.8.5 (Dynamical Curry-Howard correspondence). Through the Curry-Howard correspondence, β-reduction corresponds to eliminating cuts.

This explains the remark already made in section 2.3.3: although cut-free proofs are "simpler" in the sense that they do not contain cuts, they can be much bigger than the corresponding proofs with cuts, in a same way that executing a program can give rise to a much bigger result than the program itself (e.g. a program computing the factorial of 1000). As a direct consequence of previous theorem, we have that

Corollary 4.1.8.6. Through the Curry-Howard correspondence, typable terms in normal form correspond to cut-free proofs.

4.1.9 η-expansion. We have seen that a β-reduction step corresponds to eliminating a cut, which consists of an introduction rule followed by an elimination rule, when reading the proof from top to bottom. Similarly, an η-expansion step corresponds to introducing a "co-cut" (we are not aware of an official name for those) consisting of an elimination rule followed by an introduction rule. For instance, supposing that in some context Γ we can show $t : A \to B$, the η-expansion step

$$t \longrightarrow_\eta \lambda x^A.t\,x$$

corresponds to the following transformation of typing derivation

$$
\cfrac{
\cfrac{\vdots}{\Gamma \vdash t : A \to B}
}{\Gamma \vdash t : A \to B}
\quad\rightsquigarrow\quad
\cfrac{
\cfrac{
\cfrac{\cfrac{\vdots}{\Gamma \vdash t : A \to B}}{\Gamma, x : A \vdash t : A \to B}\ (\text{wk}) \qquad
\cfrac{}{\Gamma, x : A \vdash x : A}\ (\text{ax})
}{\Gamma, x : A \vdash t\,x : B}\ (\to_{\text{E}})
}{\Gamma \vdash \lambda x^A.t\,x : A \to B}\ (\to_{\text{I}})
$$

which, after term erasure, corresponds to the following proof transformation

$$
\cfrac{\pi}{\Gamma \vdash A \Rightarrow B}
\quad\rightsquigarrow\quad
\cfrac{
\cfrac{
\cfrac{\cfrac{\pi}{\Gamma \vdash A \Rightarrow B}}{\Gamma, A \vdash A \Rightarrow B}\ (\text{wk}) \qquad
\cfrac{}{\Gamma, A \vdash A}\ (\text{ax})
}{\Gamma, A \vdash B}\ (\Rightarrow_{\text{E}})
}{\Gamma \vdash A \Rightarrow B}\ (\Rightarrow_{\text{I}})
$$

4.1.10 Confluence. Recall from section 3.4 that β-reduction of λ-terms is confluent. By theorem 4.1.8.3, we can immediately extend this result to typable terms:

Theorem 4.1.10.1 (Confluence)*.* The β-reduction is confluent on typable terms (in some fixed context): given typable terms t, u_1 and u_2 such that $t \xrightarrow{*}_\beta u_1$ and $t \xrightarrow{*}_\beta u_2$, there exists a typable term v such that $u_1 \xrightarrow{*}_\beta v$ and $u_2 \xrightarrow{*}_\beta v$.

4.2 Strong normalization

4.2.1 A normalization strategy. We have seen in theorem 4.1.8.5 that, under the Curry-Howard correspondence, β-reduction corresponds to cut elimination. Since, in theorem 2.3.3.1, we have established that every proof reduces to a cut-free proof, this means that every typable term β-reduces to a term in normal form. More precisely, the proof produces a strategy to reduce a term to a normal form: we can reduce a β-redex $(\lambda x.t)\,u$ whenever t and u do not contain β-redexes. In fact, the proof only depends on the hypothesis that u does not contain β-redexes, and we have to suppose this because those redexes could be "duplicated" during the reduction, making it unclear that it will terminate. For instance, writing $\mathsf{I} = \lambda x.x$ for the identity, with $t = \lambda y.yxx$ and $u = \mathsf{II}$, we have the following reductions:

$$
\begin{array}{ccc}
(\lambda xy.yxx)(\mathsf{II}) & \longrightarrow & (\lambda xy.yxx)\,\mathsf{I} \\
\downarrow & & \downarrow \\
\lambda y.y(\mathsf{II})(\mathsf{II}) \longrightarrow \lambda y.y\,\mathsf{I}\,(\mathsf{II}) \longrightarrow & \lambda y.y\,\mathsf{II}
\end{array}
$$

We see that the redex $\mathsf{II} \longrightarrow_\beta \mathsf{I}$ on the top line has become two redexes in the bottom line: this is because the term $\lambda xy.yxx$ contains the variable x twice and the vertical reduction will thus cause the term substituted for x to be duplicated. Following the terminology introduced in section 3.5.1, what theorem 2.3.3.1 establishes is thus that the innermost reduction strategies, such as call-by-value, terminate for typable λ-terms.

4.2.2 Strong normalization. We would now like to show a stronger result called *strong normalization*: every typable term is strongly normalizing. This mean that, starting from a given typable term t, we will always end up with a normal form after a finite number of steps, whichever way we chose to reduce it, see section 3.2.6. We show below a proof based on "reducibility candidates" which is due to Tait [Tai75] and later refined by Girard, see [Gir89, Chapter 6]. Before entering the details of this subtle proof, let us first explain why the naive ideas for a proof do not work.

Failure of the naive proof. A first attempt to show the result would consist in showing that, for any derivable sequent $\Gamma \vdash t : A$, the term t is strongly normalizing by induction on the derivation of the sequent.

– For the rule (ax), this is obvious since a variable is strongly normalizing (it is even a normal form).

– For the rule ($\rightarrow_I$), we have to show that a term $\lambda x.t$ is strongly normalizing knowing that t is strongly normalizing. A sequence of reductions starting from $\lambda x.t$ is of the form $\lambda x.t \longrightarrow_\beta \lambda x.t_1 \longrightarrow_\beta \lambda x.t_2 \longrightarrow_\beta \ldots$ with $t \longrightarrow_\beta t_1 \longrightarrow_\beta t_2 \longrightarrow_\beta \ldots$, and is thus finite since t is strongly normalizing by induction hypothesis.

– For the rule ($\rightarrow_E$), we have to show that a term $t\,u$ is strongly normalizing knowing that both t and u are strongly normalizing. However, a reduction in $t\,u$ is not necessarily generated by a reduction in t or in u in the case where t is an abstraction, and we cannot conclude.

If we try to identify the cause of the failure, we see that we do not really use the fact that the terms are typable in the last case. We are left proving that if t and u are normalizable then $t\,u$ is normalizable, and there is a counter-example to that, already encountered in section 3.2.6: take $t = \lambda x.xx$ and $u = \lambda x.xx$, both are strongly normalizable, but $t\,u$ is not since it leads to an infinite sequence of reductions. This however is not a counter-example to the strong normalizability property, because $\lambda x.xx$ cannot be typed, but we have no easy way of exploiting this fact.

Reducibility candidates. Instead, we now take an "optimistic" approach and, given a type A, we define a set R_A of terms, called the *reducibility candidates* at A, which are terms such that

(i) for every term t such that $\Gamma \vdash t : A$ is derivable, we have $t \in R_A$,

(ii) a term t in R_A is "obviously" strongly normalizing.

Which will allow us to immediately conclude, once we have shown those properties.

The definition is performed by induction on the type A by

– for a type variable X, R_X is the set of all strongly normalizable terms t,

– for an arrow type $A \rightarrow B$, $R_{A \rightarrow B}$ is the set of terms t such that for every $u \in R_A$, we have $t\,u \in R_B$.

In the first case, we have not been particularly subtle: we wanted a set of strongly normalizable terms which contains all the terms of type X, and we simply took all strongly normalizable terms. However, in the second case, we have crafted our definition to avoid the previous problem: in the case of the rule $(\to_\mathrm{E})$, it will be obvious how to deduce, given $t \in R_{A \to B}$ and $u \in R_A$, that $t\,u \in R_B$. However, it is not obvious that every term in $R_{A \to B}$ is strongly normalizing and we will have to prove that. A term is said to be *reducible* when it belongs to a set of reducibility candidates R_A for some type A.

We begin by showing that every term $t \in R_A$ is strongly normalizing by induction on the type A, but in order to do so we need to strengthen the induction hypothesis and show together additional properties on A. A term is *neutral* when it is not an abstraction; in other words, a neutral term is of the form $t\,u$ or x.

Proposition 4.2.2.1. Given a type A and a term t, we have

(CR1) if $t \in R_A$ then t is strongly normalizing,

(CR2) if $t \in R_A$ and $t \longrightarrow_\beta t'$ then $t' \in R_A$,

(CR3) if t is neutral, and $t \longrightarrow_\beta t'$ implies $t' \in R_A$, then $t \in R_A$.

Proof. Consider a term t. We show simultaneously the three properties by induction on A. In the base case, the type A is a type variable X.

(CR1) If $t \in R_X$ then it is strongly normalizable by definition of R_X.

(CR2) Suppose that $t \in R_X$ (i.e. t is strongly normalizing) and $t \longrightarrow_\beta t'$. Every sequence of reductions $t' \longrightarrow_\beta \ldots$ starting from t' can be extended as a sequence of reductions $t \longrightarrow_\beta t' \longrightarrow_\beta \ldots$ starting from t, and is thus finite. Therefore t' is strongly normalizing and thus belongs to R_X.

(CR3) Suppose that t is neutral and such that for every term t' such that $t \longrightarrow_\beta t'$ we have $t' \in R_X$. A sequence of reductions $t \longrightarrow_\beta t' \longrightarrow_\beta \ldots$ starting from t is such that $t' \in R_X$, and is thus finite. Therefore $t \in R_X$.

Consider the case of an arrow type $A \to B$.

(CR1) Suppose that $t \in R_{A \to B}$, i.e. for every $u \in R_A$ we have $t\,u \in R_B$. A variable x is neutral and a normal form and thus belongs to R_A by (CR3). By definition of $R_{A \to B}$, we have $t\,x \in R_B$. Any sequence of reductions $t \longrightarrow_\beta t' \longrightarrow_\beta \ldots$ induces a sequence of reductions $t\,x \longrightarrow_\beta t'\,x \longrightarrow_\beta \ldots$ and is thus finite by (CR1) on B. Thus t is strongly normalizing.

(CR2) Suppose that $t \in R_{A \to B}$ and $t \longrightarrow_\beta t'$. Given a term $u \in R_A$, by definition of $R_{A \to B}$, we have $t\,u \in R_B$. Since $t\,u \longrightarrow_\beta t'\,u$, by (CR2) on B, we have $t'\,u \in R_B$. Therefore $t' \in R_{A \to B}$.

(CR3) Suppose that t is neutral and such that, for every term t' with $t \longrightarrow_\beta t'$, we have $t' \in R_{A \to B}$. Suppose given a term $u \in R_A$. By (CR1) on A, the term u is strongly normalizing and we can show that $t\,u \in R_B$ for every term $u \in R_A$ by well-founded induction on u (theorem A.3.2.1). Since t is neutral, the term $t\,u$ can only reduce in two ways.

   – If $t\,u \longrightarrow_\beta t'\,u$ then $t'\,u \in R_B$ because, by hypothesis, we have $t' \in R_{A\to B}$.

   – If $t\,u \longrightarrow_\beta t\,u'$ with $u \longrightarrow_\beta u'$ then $u' \in R_A$ by (CR2) on A and, by induction hypothesis on u, we have $t\,u' \in R_B$.

Therefore, by (CR3) on B, we have $t\,u \in R_B$. We conclude that $t \in R_{A\to B}$. $\qquad\square$

We now hope to be able to show that for every derivable sequent $\Gamma \vdash t : A$, we have $t \in R_A$, by induction on A. The case (ax) is easily handled (we have seen in the previous proof that variables belong to all sets R_A) and the case ($\to_{\mathrm{E}}$) is immediate by definition of $R_{A\to B}$. However, the case of ($\to_{\mathrm{I}}$) does not got through: from the hypothesis $t \in R_B$, we would need to deduce that $\lambda x.t \in R_{A\to B}$, i.e. that $(\lambda x.t)\,u \in R_B$ for every $u \in R_A$. Since we have $(\lambda x.t)\,u \longrightarrow_\beta t[u/x]$, this suggests proving by induction that $t[u/x] \in R_A$ instead of $t \in R_A$ (which is a particular case since $t = t[x/x]$) or, even more generally, lemma 4.2.2.3 below. We begin by the following lemma, which is used in its proof.

Lemma 4.2.2.2. Suppose given a term t such that $t[u/x] \in R_B$ for every term $u \in R_A$. Then $\lambda x^A.t \in R_{A\to B}$.

Proof. We have seen that $x \in R_A$ by (CR3) and thus $t = t[x/x]$ belongs to R_B. Given $u \in R_A$, we have to show $(\lambda x^A.t)u \in R_B$. By (CR1), the terms t and u are strongly normalizing. We can thus show $(\lambda x^A.t)u \in R_B$ by induction on the pair (t, u). The term $(\lambda x^A.t)u$ can either reduce to

   – $t[u/x]$, which is in R_B by hypothesis,

   – $(\lambda x^A.t')u$ with $t \longrightarrow_\beta t'$, which is in R_B by induction hypothesis,

   – $(\lambda x^A.t)u'$ with $u \longrightarrow_\beta u'$, which is in R_B by induction hypothesis.

In every case, the neutral term $(\lambda x^A.t)u$ reduces to a term in R_B and therefore belongs to R_B by (CR3). $\qquad\square$

Lemma 4.2.2.3. Suppose given a term t such that $\Gamma \vdash t : A$ is derivable for some context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and type A. Then, for every terms $t_i \in R_{A_i}$, for $1 \leqslant i \leqslant n$, we have $t[t_1/x_1, \ldots, t_n/x_n] \in R_A$.

Proof. We write $t[t_*/x_*]$ for the above substitution, and show the result by induction on t. By induction on the derivation of $\Gamma \vdash t : A$.

  – If the last rule is

$$\frac{}{\Gamma \vdash x_i : A_i}\ (\mathrm{ax})$$

then, for every terms $t_i \in R_{A_i}$, we have $t[t_*/x_*] = t_i \in R_{A_i}$.

  – If the last rule is

$$\frac{\Gamma \vdash u : A \to B \qquad \Gamma \vdash v : A}{\Gamma \vdash u\,v : B}\ (\to_{\mathrm{E}})$$

then, for every terms $t_i \in R_{A_i}$, by induction hypothesis, we have $u[t_*/x_*] \in R_{A\to B}$ and $v[t_*/x_*] \in R_A$, and therefore we can conclude $t[t_*/x_*] = (u[t_*/x_*])(v[t_*/x_*]) \in R_B$.

– If the last rule is

$$\frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \lambda x.u : A \to B} \ (\to_{\mathrm{I}})$$

then, by induction hypothesis, for every terms $t_i \in R_{A_i}$ and for every term $v \in R_A$, we have $u[t_*/x_*][v/x] = u[t_*/x_*, v/x] \in R_B$. Therefore, by lemma 4.2.2.2, we have $t[t_*/x_*] = \lambda x.(u[t_*/x_*]) \in R_{A \to B}$. $\qquad\square$

Proposition 4.2.2.4 (Adequacy)*.* Given a term t such that $\Gamma \vdash t : A$ is derivable, we have $t \in R_A$.

Proof. We write $\Gamma = x_1 : A_1, \ldots, x_n : A_n$. A variable being neutral and in normal form, by (CR3), we have $x_i \in R_{A_i}$ for every index i. Therefore, by lemma 4.2.2.3, $t = t[x_*/x_*] \in R_A$. $\qquad\square$

Theorem 4.2.2.5 (Strong normalization)*.* Every typable term t is strongly normalizing.

Proof. By proposition 4.2.2.4, the term t is reducible, and thus strongly normalizing by (CR1). $\qquad\square$

One of the remarkable strengths of this approach is that is generalizes well to usual extensions of simply typed λ-calculus, see section 4.3.7.

Remark 4.2.2.6. There are many possible variants on the definition of reducibility candidates, see [Gal89]. The version presented here has the advantage of being simple to define and leads to simple proofs. One of its drawbacks is that the λ-terms of R_A are not necessarily of type A (for instance, when $A = X$ any strongly normalizable term belongs to R_A by definition). We can however define a "typed variant" of reducibility candidates, by defining sets $R_{\Gamma \vdash A}$, indexed by both a context Γ and a type A, by induction on A by

– for a type variable X, $R_{\Gamma \vdash X}$ is the set of strongly normalizable terms t such that $\Gamma \vdash t : X$ is derivable,

– for an arrow type $A \to B$, $R_{\Gamma \vdash A \to B}$ is the set of terms t such that $\Gamma \vdash t : A \to B$ is derivable and for every $u \in R_{\Gamma \vdash A}$, we have $t\,u \in R_{\Gamma \vdash B}$.

The expected adaptation of the above properties hold in this context, see the formalization proposed in section 7.5.2. In particular, the variant of proposition 4.2.2.4 ensures that every term t such that $\Gamma \vdash t : A$ is derivable belongs to $R_{\Gamma \vdash A}$; conversely, one easily shows by induction on A that every term t of $R_{\Gamma \vdash A}$ is such that $\Gamma \vdash t : A$ is derivable. With this formulation, it thus turns out that reducibility candidates are simply a complicated way of defining

$$R_{\Gamma \vdash A} = \{t \mid \Gamma \vdash t : A \text{ is derivable}\}$$

However, the way the definition is formulated allows to perform the proofs by induction!

4.2.3 First consequences. We shall now present some easy consequences of the strong normalization theorem.

Non-typable terms. A first consequence of the strong normalization theorem 4.2.2.5 (or rather its contrapositive) is that there are terms which are not typable. For instance, the λ-term $\Omega = (\lambda x.xx)(\lambda x.xx)$ is not typable because it is not terminating, see section 3.2.6.

Termination of cut elimination. By theorem 4.1.8.5, cut-elimination in the implicational fragment of natural deduction corresponds to β-reduction. Since for typable terms β-reduction is always terminating (theorem 4.2.2.5), we have shown

Theorem 4.2.3.1. The cut elimination procedure of section 2.3.3 always terminates (on a cut-free proof), whichever strategy we choose to eliminate the cuts.

4.2.4 Deciding convertibility. In practice, the most important consequence of the strong normalization theorem is that it provides us with an algorithm to decide the β-convertibility of typable λ-terms t and u. Namely, suppose that we start reduce t:

$$t = t_0 \longrightarrow_\beta t_1 \longrightarrow_\beta t_2 \longrightarrow_\beta \ldots$$

This means that we start from t, reduce it to a term t_1, then reduce t_1 to a term t_2, and so on. Note that we do not impose anything on the way t_{i+1} is constructed from t_i: any reduction strategy would be acceptable. By theorem 4.2.2.5, such a sequence cannot be infinite, which means that this process will eventually give rise to a term t_n which cannot be reduced: t_n is a normal form. This shows that there exists a term in normal form $\hat{t}$ such that $t \overset{*}{\longrightarrow}_\beta \hat{t}$. Similarly, u admits a normal form $\hat{u}$. Clearly, t and u are β-convertible if and only if $\hat{t}$ and $\hat{u}$ are β-convertible. By proposition 3.4.4.3, this is the case if and only if $\hat{t}$ and $\hat{u}$ are equal:

$$
\begin{array}{ccc}
t & \overset{?}{\underset{*}{\longleftrightarrow}} & u \\
{\scriptstyle *}\downarrow & & \downarrow{\scriptstyle *} \\
\hat{t} & \underset{?}{\cdots\cdots} & \hat{u}
\end{array}
$$

We have thus reduced the problem of deciding whether two terms are convertible to deciding whether two terms are equal, which is easily done. Using the functions defined in section 3.5, the following function `eq` tests for the β-equivalence of two λ-terms which are supposed to be typable:

```
let eq t u = (normalize t) = (normalize u)
```

Remark 4.2.4.1. In fact, even if we do not suppose that the terms t and u given as input of the above function are typable, it is still correct in the sense that

- if it answers `true` then t and u are convertible, and

- if it answers `false` then t and u are not convertible.

However, there is now a third possibility: nothing guarantees that the normalization of t or u will terminate. This means that the procedure will not provide a result in such a case. As such it is not an algorithm, since, by convention, those should terminate on every input.

4.2.5 Weak normalization. The strong normalization theorem is indeed strong: it shows that, starting from a typable term, whichever way we chose to reduce a typable term, we will eventually end up with a normal form. In practice, however, we care about less than this: when implementing reduction and normalization, we implement a particular reduction strategy (see section 3.5.1), and all we want to know is that this particular strategy will end up with a normal form.

In particular, when this reduction strategy is the call-by-value strategy, which is by far the most common one, a much simplified version of the above argument can be used to show that every closed typable term is terminating according to the chosen strategy, see [Pie02, Chapter 12]. In the following, we write $t \longrightarrow u$ to indicate that t reduces to u according to the call-by-value strategy. An important point about this strategy is that it is *deterministic* in the sense that if $t \longrightarrow u$ and $t \longrightarrow u'$ then $u = u'$. Because of this, strong and weak normalization coincide for the strategy, and we simply speak of *normalizing* terms.

We define sets R_A of λ-terms by induction on A by

- $t \in R_X$ if $\vdash t : X$ is derivable and t is normalizing,

- $t \in R_{A \to B}$ if $\vdash t : A \to B$ is derivable, t is normalizing and $t\, u \in R_B$ for every $u \in R_A$.

Note that contrarily to section 4.2.2, by lemma 4.1.5.4, the sets R_A contain only closed terms of type A: although this is not necessary, it is nice to see that candidates for a type A only need to involve terms of this type. In section 4.2.2, we have been using the following property of reduction when showing the properties of reducibility candidates in proposition 4.2.2.1:

Lemma 4.2.5.1. If $t \longrightarrow t'$ and t is normalizing then t' is.

Proof. An infinite sequence of reductions $t' \longrightarrow \ldots$ starting from t' can be extended as one $t \longrightarrow t' \longrightarrow \ldots$ starting from t, i.e. if t' not strongly normalizing then t is not either. We conclude by contraposition. □

A consequence of the determinism of the strategy is that the converse of the above lemma now also holds:

Lemma 4.2.5.2. If $t \longrightarrow t'$ and t' is strongly normalizing then t is.

Proof. By determinism, an infinite sequence of reductions starting from t is necessarily of the form $t \longrightarrow t' \longrightarrow \ldots$, and thus induces one starting from t'. □

Remark 4.2.5.3. Again, this property would not be true with the relation $\longrightarrow_\beta$, which is not deterministic. For instance, we have $(\lambda x.y)\Omega \longrightarrow_\beta y$, where $(\lambda x.y)\Omega$ is not strongly normalizing but y is.

We can now easily show variants of the properties of proposition 4.2.2.1. Note that the proof is greatly simplified because we do not need to prove them all at once.

Lemma 4.2.5.4 (CR1). If $t \in R_A$ then it is strongly normalizing.

Proof. By induction on A, immediate by definition of R_A. □

Lemma 4.2.5.5 (CR2). If $t \in R_A$ and $t \longrightarrow t'$ then $t' \in R_A$.

Proof. By induction on the type A.

– Suppose that $t \in R_X$. Then t' has type A by subject reduction theorem 4.1.8.3 and t' is normalizing by lemma 4.2.5.1. Thus $t' \in R_X$.

– Suppose that $t \in R_{A \to B}$. Then t' has type A by theorem 4.1.8.3 and is normalizing by lemma 4.2.5.1. Given $u \in R_A$, we have $t\,u \in R_B$ by definition of $R_{A \to B}$, and $t\,u \longrightarrow t'\,u$ because the reduction strategy is call-by-value, and thus $t'\,u \in R_B$ by induction hypothesis. Thus $t' \in R_{A \to B}$. □

The last one uses lemma 4.2.5.2 and thus relies on the fact that we have a deterministic reduction:

Lemma 4.2.5.6 (CR3)*.* If t has type A, $t \longrightarrow t'$ and $t' \in R_A$ then $t \in R_A$.

Proof. By induction on the type A.

– Suppose that $t' \in R_X$. Then $t \in R_X$ by lemma 4.2.5.2.

– Suppose that $t' \in R_{A \to B}$. Then t' is strongly normalizing, and thus also t by lemma 4.2.5.2. Given $u \in R_A$, we have $t'\,u \in R_B$ by definition of $R_{A \to B}$, and $t\,u \longrightarrow t'\,u$ because the reduction strategy is call-by-value, and thus $t\,u \in R_B$ by induction hypothesis. Thus $t \in R_{A \to B}$. □

We can then show the following:

Lemma 4.2.5.7 (Adequacy)*.* Suppose given a term t such that $\Gamma \vdash t : A$ is derivable for some context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and type A. Then, for every terms $t_i \in R_{A_i}$, for $1 \leqslant i \leqslant n$, we have $t[t_1/x_1, \ldots, t_n/x_n] \in R_A$.

Proof. The result is shown by induction on the derivation of $\Gamma \vdash t : A$.

– If the last rule is

$$\frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \lambda x.u : A \to B} \; (\to_{\mathrm{I}})$$

then, by induction hypothesis, for every terms $t_i \in R_{A_i}$ and for every term $v \in R_A$, we have $u[t_*/x_*][v/x] = u[t_*/x_*, v/x] \in R_B$. Since $v \in R_A$, it is normalizing and there is a reduction $v \stackrel{*}{\longrightarrow} \hat{v}$ to some normal form $\hat{v}$, and we have $\hat{v} \in R_A$ by (CR2). In the call-by-value reduction strategy, we have

$$(\lambda x.u)\,v \stackrel{*}{\longrightarrow} (\lambda x.u)\,\hat{v} \longrightarrow u[\hat{v}/x]$$

thus,

$$(\lambda x.u[t_*/x_*])\,v \stackrel{*}{\longrightarrow} u[t_*/x_*, \hat{v}/x]$$

where the term on the right belongs to R_B by induction hypothesis, and therefore the term on the left as well by (CR3). Since this holds for any term $v \in R_A$, we have shown $t[t_*/x_*] = \lambda x.(u[t_*/x_*]) \in R_{A \to B}$.

Other cases are handled as in lemma 4.2.2.3. □

Finally, we can deduce

Theorem 4.2.5.8*.* Given a term t, if there is a type A such that $\vdash t : A$ is derivable then t is normalizing.

Proof. Suppose that $\vdash t : A$ holds. By lemma 4.2.5.7, we have that $t \in R_A$. Thus t is normalizing by lemma 4.2.5.4. □

The call-by-value reduction strategy is *complete* in the following sense: for every term t, if there is a term u such that $t \longrightarrow_\beta u$ then there is a term u' such that $t \longrightarrow u'$. In other words, our strategy can reduce any β-reducible term. We thus deduce that

Theorem 4.2.5.9 (Weak normalization)*.* Every typable term is weakly normalizing.

A formalization of these properties is provided in section 7.5.2.

4.3 Other connectives

Up to now, for simplicity, we have been limiting ourselves to types built using arrows as the only connective. The Curry-Howard correspondence would be sad if it stopped there: it actually extends to other usual connectives. For instance, the product of two types corresponds to taking the conjunction of the two corresponding formulas. Other cases of the correspondence are given in the following table:

Typing			Logic	
function	$\to$	$\Rightarrow$	implication	
product	$\times$	$\wedge$	conjunction	
unit	1	$\top$	truth	
coproduct	$+$	$\vee$	disjunction	
empty	0	$\bot$	falsity	

In order to study this, we will now consider types generated by the following syntax:
$$A, B ::= X \mid A \to B \mid A \times B \mid 1 \mid A + B \mid 0$$

For each of those connectives, we add a connective between types, as well as new constructions to the λ-calculus which correspond to introduction and elimination rules, and the full syntax for λ-terms will be

$$
\begin{aligned}
t, u ::= {}& x \mid t\,u \mid \lambda x^A.t \\
& \mid \langle t, u \rangle \mid \pi_l(t) \mid \pi_r(t) \mid \langle\rangle \\
& \mid \iota_l^A(t) \mid \iota_r^A(t) \mid \mathrm{case}(t, x \mapsto u, y \mapsto v) \mid \mathrm{case}^A(t)
\end{aligned}
$$

Moreover, each such connective will give rise to typing rules and the full list of rules is given in figure 4.3. In addition, we need to add new rules for β-reduction, which correspond to cut elimination for the new rules (see section 4.1.8), resulting in the rules of figure 4.4, and η-expansion rules which correspond to introducing "co-cuts" (see section 4.1.9). We now gradually introduce each of those.

Most of the important theorems extend to the λ-calculus with these new added constructors and types, although we will not detail these:

– confluence (theorem 3.4.3.7),

– subject reduction (theorem 4.1.8.3),

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \ (\text{ax})$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B} \ (\to_{\text{E}}) \qquad\qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \to B} \ (\to_{\text{I}})$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_{\text{l}}(t) : A} \ (\times_{\text{E}}^{\text{l}}) \quad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_{\text{r}}(t) : B} \ (\times_{\text{E}}^{\text{r}}) \quad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} \ (\times_{\text{I}})$$

$$\frac{}{\Gamma \vdash \langle \rangle : 1} \ (1_{\text{I}})$$

$$\frac{\Gamma \vdash t : A + B \qquad \Gamma, x : A \vdash u : C \qquad \Gamma, y : B \vdash v : C}{\Gamma \vdash \text{case}(t, x \mapsto u, y \mapsto v) : C} \ (+_{\text{E}})$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \iota_{\text{l}}^B(t) : A + B} \ (+_{\text{I}}^{\text{l}}) \quad \frac{\Gamma \vdash t : B}{\Gamma \vdash \iota_{\text{r}}^A(t) : A + B} \ (+_{\text{I}}^{\text{r}})$$

$$\frac{\Gamma \vdash t : 0}{\Gamma \vdash \text{case}^A(t) : A} \ (0_{\text{E}})$$

Figure 4.3: Typing rules for λ-calculus with products and sums.

β-reduction rules:

$$(\lambda x.t)\, u \longrightarrow_\beta t[u/x]$$
$$\pi_{\mathrm{l}}(\langle t, u \rangle) \longrightarrow_\beta t$$
$$\pi_{\mathrm{r}}(\langle t, u \rangle) \longrightarrow_\beta u$$
$$\mathrm{case}(\iota_{\mathrm{l}}^{B}(t), x \mapsto u, y \mapsto v) \longrightarrow_\beta u[t/x]$$
$$\mathrm{case}(\iota_{\mathrm{r}}^{A}(t), x \mapsto u, y \mapsto v) \longrightarrow_\beta v[t/y]$$

Commuting reduction rules:

$$\mathrm{case}^{A \to B}(t)\, u \longrightarrow_\beta \mathrm{case}^{B}(t)$$
$$\pi_{\mathrm{l}}(\mathrm{case}^{A \times B}(t)) \longrightarrow_\beta \mathrm{case}^{A}(t)$$
$$\pi_{\mathrm{r}}(\mathrm{case}^{A \times B}(t)) \longrightarrow_\beta \mathrm{case}^{B}(t)$$
$$\mathrm{case}(\mathrm{case}^{A+B}(t), x \mapsto u, y \mapsto v) \longrightarrow_\beta \mathrm{case}^{C}(t)$$
$$\mathrm{case}^{A}(\mathrm{case}^{0}(t)) \longrightarrow_\beta \mathrm{case}^{A}(t)$$
$$\mathrm{case}(t, x \mapsto u, y \mapsto v)\, w \longrightarrow_\beta \mathrm{case}(t, x \mapsto u\, w, y \mapsto v\, w)$$
$$\pi_{\mathrm{l}}(\mathrm{case}(t, x \mapsto u, y \mapsto v)) \longrightarrow_\beta \mathrm{case}(t, x \mapsto \pi_{\mathrm{l}}(u), y \mapsto \pi_{\mathrm{l}}(v))$$
$$\pi_{\mathrm{r}}(\mathrm{case}(t, x \mapsto u, y \mapsto v)) \longrightarrow_\beta \mathrm{case}(t, x \mapsto \pi_{\mathrm{r}}(u), y \mapsto \pi_{\mathrm{r}}(v))$$
$$\mathrm{case}^{C}(\mathrm{case}(t, x \mapsto u, y \mapsto v)) \longrightarrow_\beta \mathrm{case}(t, x \mapsto \mathrm{case}^{C}(u), y \mapsto \mathrm{case}^{C}(v))$$
$$\mathrm{case}(\mathrm{case}(t, x \mapsto u, y \mapsto v), x' \mapsto u', y' \mapsto v')$$
$$\longrightarrow_\beta$$
$$\mathrm{case}(t, x \mapsto \mathrm{case}(u, x' \mapsto u', y' \mapsto v'), y \mapsto \mathrm{case}(v, x' \mapsto u', y' \mapsto v'))$$

Figure 4.4: Reduction rules for λ-calculus with products and sums.

– strong normalization (theorem 4.2.2.5),

– Curry-Howard correspondence (theorems 4.1.7.1 and 4.1.8.5).

4.3.1 Products. In order to accommodate products, we add the construction

$$A \times B$$

to the syntax of our types, which corresponds to the product of two types A and B. We also extend λ-terms with three new constructions:

$$t, u ::= \ldots \mid \langle t, u \rangle \mid \pi_l(t) \mid \pi_r(t)$$

where

– $\langle t, u \rangle$ is the pair of two λ-terms t and u,

– $\pi_l(t)$ takes the first component of the λ-term t,

– $\pi_r(t)$ takes the second component of the λ-term t.

We add three new typing rules to our system, one for each of the newly added constructors:

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_l(t) : A} \; (\times_E^l) \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_r(t) : B} \; (\times_E^r) \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} \; (\times_I)$$

The first one states that if t is a term, which is a pair consisting of an element of A and an element of B, then the term $\pi_l(t)$, obtained by taking its first projection, has type A. The second rule is similar. The last rule establishes that if t is of type A and u is of type B then the pair $\langle t, u \rangle$ is of type $A \times B$.

If we apply our term erasing procedure of section 4.1.7, and replace the symbols $\times$ by $\wedge$, we recover the rules for conjunction:

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \; (\wedge_E^l) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \; (\wedge_E^r) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \; (\wedge_I)$$

This means that our extension of simply typed λ-calculus is compatible with the Curry-Howard correspondence (theorem 4.1.7.1).

Recall that the cut-elimination rules for conjunction consist in the following two cases:

$$\frac{\dfrac{\dfrac{\pi}{\Gamma \vdash A} \qquad \dfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B} \; (\wedge_I)}{\Gamma \vdash A} \; (\wedge_E^l) \quad \rightsquigarrow \quad \dfrac{\pi}{\Gamma \vdash A}$$

$$\frac{\dfrac{\dfrac{\pi}{\Gamma \vdash A} \qquad \dfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B} \; (\wedge_I)}{\Gamma \vdash B} \; (\wedge_E^r) \quad \rightsquigarrow \quad \dfrac{\pi'}{\Gamma \vdash B}$$

By the Curry-Howard correspondence, they correspond to the following transformations of typing derivations:

$$\cfrac{\cfrac{\cfrac{\dfrac{\pi}{\Gamma \vdash t : A} \qquad \dfrac{\pi'}{\Gamma \vdash u : B}}{\Gamma \vdash \langle t, u \rangle : A \times B}\ (\times_{\mathrm{I}})}{\Gamma \vdash \pi_{\mathrm{l}}(\langle t, u \rangle) : A}\ (\times_{\mathrm{E}}^{\mathrm{l}})}{} \qquad \rightsquigarrow \qquad \dfrac{\pi}{\Gamma \vdash t : A}$$

$$\cfrac{\cfrac{\cfrac{\dfrac{\pi}{\Gamma \vdash t : A} \qquad \dfrac{\pi'}{\Gamma \vdash u : B}}{\Gamma \vdash \langle t, u \rangle : A \times B}\ (\times_{\mathrm{I}})}{\Gamma \vdash \pi_{\mathrm{r}}(\langle t, u \rangle) : B}\ (\times_{\mathrm{E}}^{\mathrm{r}})}{} \qquad \rightsquigarrow \qquad \dfrac{\pi'}{\Gamma \vdash u : A}$$

which indicate that the reduction rules associated to the new connectives should be

$$\pi_{\mathrm{l}}(\langle t, u \rangle) \longrightarrow_{\beta} t \qquad\qquad \pi_{\mathrm{r}}(\langle t, u \rangle) \longrightarrow_{\beta} u$$

as expected: taking the first component of a pair $\langle t, u \rangle$ returns t, and similarly for the second component.

Finally, the η-expansion rule corresponds to the following transformation of the proof derivation:

$$\dfrac{\pi}{\Gamma \vdash t : A \times B} \qquad \rightsquigarrow \qquad \cfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash t : A \times B}}{\Gamma \vdash \pi_{\mathrm{l}}(t) : A}\ (\times_{\mathrm{E}}^{\mathrm{l}}) \qquad \dfrac{\dfrac{\pi}{\Gamma \vdash t : A \times B}}{\Gamma \vdash \pi_{\mathrm{r}}(t) : B}\ (\times_{\mathrm{E}}^{\mathrm{r}})}{\Gamma \vdash \langle \pi_{\mathrm{l}}(t), \pi_{\mathrm{r}}(t) \rangle : A \times B}\ (\times_{\mathrm{I}})$$

It should thus consist in the rule

$$t \longrightarrow_{\eta} \langle \pi_{\mathrm{l}}(t), \pi_{\mathrm{r}}(t) \rangle$$

which states some form of "extensionality" of products: a term which is a product should be the same as the pair consisting of its components.

Alternative formulations. Alternative formulations are possible for those connectives. Instead of taking the first and second projection of some term t, i.e. $\pi_{\mathrm{l}}(t)$ and $\pi_{\mathrm{r}}(t)$, we could simply add to our calculus the first and second projection operators $\pi_{\mathrm{l}}^{A,B}$ and $\pi_{\mathrm{r}}^{A,B}$, whose associated typing rules would be

$$\dfrac{}{\Gamma \vdash \pi_{\mathrm{l}}^{A,B} : A \times B \to A} \qquad\qquad \dfrac{}{\Gamma \vdash \pi_{\mathrm{r}}^{A,B} : A \times B \to B}$$

This would correspond to an approach using combinators, also called Hilbert-style, see section 4.5.

Another alternative, could be to add a constructor

$$\mathrm{unpair}(t, xy \mapsto u)$$

which would corresponds to OCaml's

```
let (x,y) = t in u
```

It binds the two components of a pair t as x and y in u. The corresponding typing rule would be

$$\frac{\Gamma \vdash t : A \times B \qquad \Gamma, x : A, y : B \vdash u : C}{\Gamma \vdash \mathrm{unpair}(t, xy \mapsto u) : C}$$

This is the flavor of rules which has to be used when working with dependent types, see section 8.3.3. We did not use it here because, through the Curry-Howard correspondence, it corresponds to the following variant of elimination rule for conjunction

$$\frac{\Gamma \vdash A \wedge B \qquad \Gamma, A, B \vdash C}{\Gamma \vdash C} \ (\wedge_{\mathrm{E}})$$

which is not the one which is traditionally used (the main reason is that it involves a "new" formula C, whereas the usual rules $(\wedge_{\mathrm{E}}^{\mathrm{l}})$ and $(\wedge_{\mathrm{E}}^{\mathrm{r}})$ only use A and B).

Currying. An important property of product types in λ-calculus is that they are closely related to arrow types through an isomorphism called *currying*, which states that a function with two arguments is the same as a function taking a pair of arguments. More precisely, given types A, B and C, the two types

$$A \times B \to C \qquad \text{and} \qquad A \to B \to C$$

are isomorphic, see remark 4.1.7.3, where the first type is implicitly bracketed as $(A \times B) \to C$. In OCaml, this means that it is roughly the same to write a function of the form

```
let f x y = ...
```

or a function of the form

```
let f (x, y) = ...
```

More precisely, the isomorphism between the two types means that we have λ-terms which allow converting elements of on type into an element of the other type, in both directions:

$$\lambda f^{A \times B \to C}.\lambda a^A.\lambda b^B.f \langle a, b \rangle : (A \times B \to C) \to (A \to B \to C)$$
$$\lambda f^{A \to B \to C}.\lambda x^{A \times B}.f \, \pi_{\mathrm{l}}(x) \, \pi_{\mathrm{r}}(x) : (A \to B \to C) \to (A \times B \to C)$$

whose composites are both identities (up to $\beta\eta$-equivalence). Namely, writing t and u respectively for those terms, we have

$$\lambda f^{A \times B \to C}.u \, (t \, f) \overset{*}{\longrightarrow}_\beta \lambda f^{A \times B \to C}.\lambda x^{A \times B}.f \langle \pi_{\mathrm{l}}(x), \pi_{\mathrm{r}}(x) \rangle ==_\eta \lambda f^{A \times B \to C}.f$$
$$\lambda f^{A \to B \to C}.t \, (u \, f) \overset{*}{\longrightarrow}_\beta \lambda f^{A \to B \to C}.\lambda a^A.\lambda b^B.f \, a \, b ==_\eta \lambda f^{A \to B \to C}.f$$

In most programming languages (Java, C, etc), a function with two arguments would be given a type of the form $A \times B \to C$. Functional programming languages such as the OCaml tend to prefer types of the form $A \to B \to C$ because they allows partial evaluation, meaning that we can give the argument of type A without giving the other one.

4.3.2 Unit. We can add a unit type by adding a constant type

$$1$$

called *unit*. It corresponds to the `unit` type of OCaml and, through the Curry-Howard correspondence to the formula $\top$. We add a new constant λ-term $\langle\rangle$ which is the only element of 1 (up to β-equivalence), and corresponds to `()` in OCaml. The typing rule is

$$\frac{}{\Gamma \vdash \langle\rangle : 1} \ (1_\mathrm{I})$$

which corresponds to the usual rule for truth by term erasure:

$$\frac{}{\Gamma \vdash \top} \ (\top_\mathrm{I})$$

There are no β- or η-reduction rules.

4.3.3 Coproducts. For coproducts, we add the construction

$$A + B$$

on types, which represents the coproduct of the two types A and B. Intuitively, this corresponds to the set-theoretic disjoint union: an element of $A+B$ is either an element of type A or an element of type B. We add three new constructions to the syntax of λ-terms:

$$t, u, v ::= \dots \mid \mathrm{case}(t, x \mapsto u, y \mapsto v) \mid \iota_\mathrm{l}^A(t) \mid \iota_\mathrm{r}^A(t)$$

where t, u and v are terms, x and y are variables and A is a type. Since $A + B$ is the disjoint union of A and B, we should be able to see a term t of type A (resp. B) as a term of type $A + B$: this is precisely represented by the term $\iota_\mathrm{l}^B(t)$ (resp. $\iota_\mathrm{r}^A(t)$), which can be thought of as the term t "cast" into an element of type $A+B$. For this reason, ι_l^A and ι_r^A are often called the *canonical injections*. Conversely, any element of $A + B$ should either be an element of A or an element of B. This means that we should be able to construct new values by case analysis: for instance, given a term t of type $A + B$,

   – if t is of type A then we return $u(t)$,

   – if t is of type B then we return $v(t)$.

Above, u (resp. v) should be a λ-term with a distinguished free variable x (resp. y) which is to be replaced by t. In formal notation, such a case analysis is written

$$\mathrm{case}(t, x \mapsto u, y \mapsto v)$$

The symbol "$\mapsto$" is purely formal here (it indicates bound variables), and our operation takes 5 arguments t, x, u, y and v. With the above intuitions, it should be no surprise that the typing rules are

$$\frac{\Gamma \vdash t : A + B \qquad \Gamma, x : A \vdash u : C \qquad \Gamma, y : B \vdash v : C}{\Gamma \vdash \mathrm{case}(t, x \mapsto u, y \mapsto v) : C} \ (+_\mathrm{E})$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \iota_\mathrm{l}^B(t) : A + B} \ (+_\mathrm{I}^\mathrm{l}) \qquad\qquad \frac{\Gamma \vdash B}{\Gamma \vdash \iota_\mathrm{r}^A(t) : A + B} \ (+_\mathrm{I}^\mathrm{r})$$

From a Curry-Howard perspective, $+$ corresponds to disjunction $\vee$, and term erasure of the above typing rules do indeed allow us to recover the usual rules for disjunction:

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C} \ (\vee_{\mathrm{E}})$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \ (\vee_{\mathrm{I}}^{\mathrm{l}}) \qquad\qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \ (\vee_{\mathrm{I}}^{\mathrm{r}})$$

The β-reduction rules correspond to the cut-elimination step reducing

$$\frac{\dfrac{\dfrac{\pi}{\Gamma \vdash t : A}}{\Gamma \vdash \iota_{\mathrm{l}}^{B}(t) : A + B} \ (+_{\mathrm{I}}^{\mathrm{l}}) \qquad \dfrac{\pi'}{\Gamma, x : A \vdash u : C} \qquad \dfrac{\pi''}{\Gamma, y : B \vdash v : C}}{\Gamma \vdash \mathrm{case}(\iota_{\mathrm{l}}^{B}(t), x \mapsto u, y \mapsto v) : C} \ (+_{\mathrm{E}})$$

to

$$\frac{\pi'[\pi/x]}{\Gamma \vdash u[t/x] : C}$$

as well as the symmetric one, obtained by using ι_{r} instead of ι_{l}. The β-reduction rules are thus

$$\mathrm{case}(\iota_{\mathrm{l}}^{B}(t), x \mapsto u, y \mapsto v) \longrightarrow_{\beta} u[t/x]$$
$$\mathrm{case}(\iota_{\mathrm{r}}^{A}(t), x \mapsto u, y \mapsto v) \longrightarrow_{\beta} v[t/y]$$

The η-expansion rule is

$$t \longrightarrow_{\eta} \mathrm{case}(t, x \mapsto \iota_{\mathrm{l}}^{B}(t), y \mapsto \iota_{\mathrm{r}}^{A}(t))$$

In OCaml. We recall from section 1.3.2 that coproducts can be implemented in OCaml as the type

```
type ('a, 'b) coprod =
  | Left  of 'a
  | Right of 'b
```

The injections ι_{l} and ι_{r} respectively correspond to `Left` and `Right`, and the eliminator $\mathrm{case}(t, x \mapsto u, y \mapsto v)$ to

```
match t with
  | Left  x -> u
  | Right y -> v
```

Church vs Curry style. Note that if we remove the type annotations on the injections, i.e. write $\iota_{\mathrm{l}}(t)$ instead of $\iota_{\mathrm{l}}^{B}(t)$, then the typing of a λ-term is not unique anymore. Namely, the typing rules become

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \iota_{\mathrm{l}}(t) : A + B} \ (+_{\mathrm{I}}^{\mathrm{l}}) \qquad\qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \iota_{\mathrm{r}}(t) : A + B} \ (+_{\mathrm{I}}^{\mathrm{r}})$$

and, in the first rule, there is no way of guessing the type B in the conclusion from the premise (and similarly for the other rule). Similar issues happen if we remove the type annotations from abstractions, these are detailed in section 4.4.

α-conversion and substitution. The reason why we use the symbol "↦" in terms $\text{case}(t, x \mapsto u, y \mapsto v)$ is that it indicates that x is bound in u (and similarly for v), in the sense that our α-equivalence should include

$$\text{case}(t, x \mapsto u, y \mapsto v) =_\alpha \text{case}(t, x' \mapsto u[x'/x], y' \mapsto v[y'/y])$$

This also means that substitution should take care not to accidentally bind variables: the equation

$$(\text{case}(t, x \mapsto u, y \mapsto v))[w/z] = \text{case}(t[w/z], x \mapsto u[w/z], y \mapsto v[w/z])$$

is valid only when $x \notin \text{FV}(w)$ and $y \notin \text{FV}(w)$.

Such details can be cumbersome in practice when performing implementations, and we already have spent a great deal of time doing this correctly for abstractions. It is possible to use an alternative formulation of the eliminator which allows the use of abstractions, thus simplifying implementations by having abstractions being the only case where we have to be careful about capture of variables: in the construction $\text{case}(t, x \mapsto u, y \mapsto v)$, instead of having u (resp. v) be a λ-term with a distinguished free variable x (resp. y), we can directly describe it as the function $\lambda x.u$ (resp. $\lambda y.v$). Our eliminator thus now has the form

$$\text{case}(t, u, v)$$

taking three terms in argument, with associated typing rule

$$\frac{\Gamma \vdash t : A + B \qquad \Gamma \vdash u : A \to C \qquad \Gamma \vdash v : B \to C}{\Gamma \vdash \text{case}(t, u, v) : C} \ (+_\text{E})$$

and the β-reduction rules become

$$\text{case}(\iota_1^B(t), u, v) \longrightarrow_\beta u\,t \qquad\qquad \text{case}(\iota_\text{r}^A(t), u, v) \longrightarrow_\beta v\,t$$

4.3.4 Empty type. The empty type is usually written 0. We extend the syntax of λ-terms

$$t ::= \ldots \mid \text{case}^A(t)$$

by adding one eliminator $\text{case}^A(t)$ which allows us to construct an element of an arbitrary type A, provided that we have constructed an element of the empty type 0 (which we do not expect to be possible). The typing rule is thus

$$\frac{\Gamma \vdash t : 0}{\Gamma \vdash \text{case}^A(t) : A} \ (0_\text{E})$$

Through the Curry-Howard correspondence, the type 0 corresponds to the falsity formula $\bot$, and we recover the usual elimination rule

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \ (\bot_\text{E})$$

There is no β-reduction rule and the η-reduction rule is

$$\text{case}^0(t) \longrightarrow_\eta t$$

for t of type 0, which corresponds to the transformation

$$\cfrac{\cfrac{\pi}{\Gamma \vdash t : 0}}{\Gamma \vdash \mathrm{case}^0(t) : 0}\ (0_{\mathrm{E}}) \qquad \leadsto \qquad \cfrac{\pi}{\Gamma \vdash t : 0}$$

As usual, negation can be implemented $\neg A = A \Rightarrow \bot$, i.e. we could define the corresponding type $\neg A = A \to 0$.

4.3.5 Commuting conversions. As explained in section 2.3.6, when considering both conjunctions and disjunctions, usual cuts are not the only situations where we want to simplify proofs, we also want to be able to remove the commutative cuts. By the Curry-Howard correspondence, this means that when having λ-terms with both products and coproducts, we want some additional reduction rules, called *commuting conversions*, which are all listed in figure 4.4.

For instance, we have the following commutative cut

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash A \vee B} \qquad \cfrac{\pi'}{\Gamma, A \vdash C \wedge D} \qquad \cfrac{\pi''}{\Gamma, B \vdash C \wedge D}}{\Gamma \vdash C \wedge D}\ (\vee_{\mathrm{E}})}{\Gamma \vdash C}\ (\wedge_{\mathrm{E}}^1)$$

which reduces to

$$\cfrac{\cfrac{\pi}{\Gamma \vdash A \vee B} \qquad \cfrac{\cfrac{\pi'}{\Gamma, A \vdash C \wedge D}}{\Gamma, A \vdash C}\ (\wedge_{\mathrm{E}}^1) \qquad \cfrac{\cfrac{\pi''}{\Gamma, B \vdash C \wedge D}}{\Gamma, B \vdash C}\ (\wedge_{\mathrm{E}}^1)}{\Gamma \vdash C}\ (\vee_{\mathrm{E}})$$

By the Curry-Howard correspondence, this means that the typing derivation

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash t : A + B} \qquad \cfrac{\pi'}{\Gamma, x : A \vdash u : C \wedge D} \qquad \cfrac{\pi''}{\Gamma, y : B \vdash v : C \wedge D}}{\Gamma \vdash \mathrm{case}(t, x \mapsto u, y \mapsto v) : C \wedge D}\ (+_{\mathrm{E}})}{\Gamma \vdash \pi_1(\mathrm{case}(t, x \mapsto u, y \mapsto v)) : C}\ (\times_{\mathrm{E}}^1)$$

should reduce to

$$\cfrac{\cfrac{\pi}{\Gamma \vdash t : A + B} \qquad \cfrac{\cfrac{\pi'}{\Gamma, x : A \vdash u : C \times D}}{\Gamma, x : A \vdash \pi_1(u) : C}\ (\times_{\mathrm{E}}^1) \qquad \cfrac{\cfrac{\pi''}{\Gamma, y : B \vdash v : C \times D}}{\Gamma, y : B \vdash \pi_1(v) : C}\ (\times_{\mathrm{E}}^1)}{\Gamma \vdash \mathrm{case}(t, x \mapsto \pi_1(u), y \mapsto \pi_1(v)) : C}\ (+_{\mathrm{E}})$$

and thus that we should add the reduction rule

$$\pi_1(\mathrm{case}(t, x \mapsto u, y \mapsto v)) \longrightarrow_\beta \mathrm{case}(t, x \mapsto \pi_1(u), y \mapsto \pi_1(v))$$

which states that projections can "go through" case operators. Other rules are obtained similarly.

4.3.6 Natural numbers. In order to grow λ-calculus into a more full-fledged programming language, it is also possible to add basic types (integers, strings, etc.) as well as constants and functions to operate on those. In order to illustrate this, we explain here how to extend simply typed λ-calculus with natural numbers. The resulting system, called *system T*, was originally studied by Gödel [Göd58].

The types are generated by

$$A ::= X \mid A \to B \mid \text{Nat}$$

where the newly added type Nat stands for natural numbers. Terms are generated by

$$t, u, v ::= x \mid t\,u \mid \lambda x.t \mid \text{Z} \mid \text{S}(t) \mid \text{rec}(t, u, xy \mapsto v)$$

where the term Z stands for the zero constant, and $\text{S}(t)$ for the successor of a term t (supposed to be a natural number). The construction $\text{rec}(t, u, xy \mapsto v)$, known as *recursor*, allows definition of functions by induction:

– if t is 0, it returns u,

– if t is $n + 1$, it returns v where x has been replaced by n and y by the value recursively computed for n.

In OCaml. In OCaml, using `int` as representation for natural numbers, Z would correspond to `0`, $\text{S}(t)$ to `t+1` and the recursor to

```
let rec recursor t u v =
  if t = 0 then u else v (t-1) (recursor (t-1) u v)
```

Alternatively, we can represent natural numbers as the type

```
type nat = Z | S of nat
```

where Z corresponds to Z, S to S and the recursor to

```
let rec recursor t u v =
  match t with
  | Z   -> u
  | S n -> v n (recursor n u v)
```

Traditionally, addition can be defined by induction by

```
let rec add m n =
  match m with
  | Z   -> n
  | S m -> S (add m n)
```

However, it can be observed that all the induction power we usually need is already contained in the recursor, so that this can equivalently be defined as

```
let add m n = recursor m n (fun m r -> S r)
```

Similarly, multiplication can be defined as

```
let mul m n = recursor m Z (fun m r -> add r n)
```

and other traditional functions (exponentiation, Ackermann's function, etc.) are left to the reader.

There are, however, some functions that can be written using recursion in OCaml, but cannot be encoded using recursor. For instance, all functions written with the recursor are total and therefore the function

```
let rec omega n = omega n
```

cannot be implemented using it, since it never produces a result whereas the recursor always defines well-defined functions.

Rules. The typing rules for the new terms are the following ones:

$$\frac{}{\Gamma \vdash \mathrm{Z} : \mathrm{Nat}} \ (\mathrm{Z_I}) \qquad\qquad \frac{\Gamma \vdash t : \mathrm{Nat}}{\Gamma \vdash \mathrm{S}(t) : \mathrm{Nat}} \ (\mathrm{S_I})$$

$$\frac{\Gamma \vdash t : \mathrm{Nat} \qquad \Gamma \vdash u : A \qquad \Gamma, x : \mathrm{Nat}, y : A \vdash v : A}{\Gamma \vdash \mathrm{rec}(t, u, xy \mapsto v) : A} \ (\mathrm{Nat_E})$$

The reduction rules ensure that the recursor implements the primitive recursion rules:

$$\mathrm{rec}(\mathrm{Z}, u, xy \mapsto v) \longrightarrow_\beta u$$
$$\mathrm{rec}(\mathrm{S}(t), u, xy \mapsto v) \longrightarrow_\beta v[t/x, \mathrm{rec}(t, u, xy \mapsto v)/y]$$

Properties. It can be shown, see section 4.3.7, that this system is terminating and confluent. Moreover, the functions of type

$$\mathrm{Nat} \to \mathrm{Nat}$$

which can be implemented in this system are precisely the recursive functions which are provably total (in Peano Arithmetic, see section 5.2.5), i.e. recursive functions for which there is a proof that they terminate on every input. This class of functions strictly includes the primitive recursive ones, and it is strictly included in the class of total recursive functions.

4.3.7 Strong normalization. The strong normalization proof presented in section 4.2.2 for simply typed λ-calculus extends to the other connectives presented above. For instance, following [Gir89, chapter 7], let us briefly explain how to adapt the proof for a λ-calculus with products, unit and natural numbers. Types are thus generated by

$$A, B ::= X \mid A \to B \mid A \times B \mid 1 \mid \mathrm{Nat}$$

and terms by

$$t, u, v ::= x \mid \lambda x^A.t \mid t\,u \mid \langle t, u \rangle \mid \pi_\mathrm{l}(t) \mid \pi_\mathrm{r}(t) \mid \langle \rangle \mid \mathrm{Z} \mid \mathrm{S}(t) \mid \mathrm{rec}(t, u, xy \mapsto v)$$

We extend the notion of *reducibility candidate* by

$$R_X = R_1 = R_\mathrm{Nat} = \{t \mid t \text{ is strongly normalizable}\}$$
$$R_{A \to B} = \{t \mid u \in R_B \text{ implies } t\,u \in R_A\}$$
$$R_{A \times B} = \{t \mid \pi_\mathrm{l}(t) \in R_A \text{ and } \pi_\mathrm{r}(t) \in R_B\}$$

We also extend the notion of *neutral term*: a term is neutral when it is not of the following forms

$$\lambda x^A.t \qquad \langle t, u \rangle \qquad \langle \rangle \qquad \mathrm{Z} \qquad \mathrm{S}(t)$$

which correspond to the possible introduction rules in our system. With those definitions, the proofs can be performed following the same structure as in section 4.2.2.

4.4 Curry style typing

In this section, we investigate simply typed λ-calculus *à la Curry* when abstractions are of the form $\lambda x.t$ instead of $\lambda x^A.t$, i.e. we do not indicate the type of abstracted variables. A detailed presentation of this topic can be found in [Pie02, chapter 22].

4.4.1 A typing system. Curry-style typing is closer to languages such as OCaml, where we do not have to indicate the type of the arguments of a function. For simplicity, we consider functions only, i.e. types are defined by

$$A, B ::= X \mid A \to B$$

and terms are defined by

$$t, u ::= x \mid \lambda x.t \mid t\,u$$

similarly to the beginning of this chapter. The typing rules are

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \ (\mathrm{ax})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \ (\to_\mathrm{I})$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B} \ (\to_\mathrm{E})$$

This seemingly minor change of not writing types for abstractions has major consequences on the properties of typing. In particular, theorem 4.1.6.1 does not hold anymore: a given λ-term might admit multiple types. For instance, the identity λ-term admits the following types:

$$\frac{\dfrac{}{x : X \vdash x : X} \ (\mathrm{ax})}{\vdash \lambda x.x : X \to X} \ (\to_\mathrm{I}) \qquad\qquad \frac{\dfrac{}{x : Y \to Z \vdash x : Y \to Z} \ (\mathrm{ax})}{\vdash \lambda x.x : (Y \to Z) \to (Y \to Z)} \ (\to_\mathrm{I})$$

and in fact, every type of the form $A \to A$ for some type A is an admissible type for the identity.

The reason for this is that when we derive a type containing a type variable in this system, we can always replace this variable by any other type. Formally,

given types A and B and a type variable X, we write $A[B/X]$ for the type obtained from A by replacing every occurrence of X by B in A. Similarly, given a context Γ, we write $\Gamma[B/X]$ for the context where X has been replaced by B in every type. We have

Lemma 4.4.1.1. If $\Gamma \vdash t : A$ is derivable then $\Gamma[B/X] \vdash t : A[B/X]$ is also derivable for every type B and variable X.

Proof. By induction on the derivation of $\Gamma \vdash t : A$. □

For instance, since the identity admits the type $X \to X$, it also admits the same type where X has been replaced by $Y \to Z$, i.e. $(Y \to Z) \to (Y \to Z)$. The first type is "more general" than the second, in the sense that the second can be obtained by substituting type variables in the first. We will see that any term admits a type which is "most general", in the sense that it is more general than any other of its types. For instance, the most general type for identity is $X \to X$. Again, this phenomenon is not present in Church style typing, e.g. the two terms

$$\lambda x^X.x : X \to X \qquad \lambda x^{Y \to Z}.x : (Y \to Z) \to (Y \to Z)$$

are distinct: Curry is more spicy than Church.

4.4.2 Principal types. Recall from section 2.2.11 that a *substitution* σ is a function which associates a type to each type variable in $\mathcal{X}$. Its domain $\mathrm{dom}(\sigma)$ is the set of type variables

$$\mathrm{dom}(\sigma) = \{X \in \mathcal{X} \mid \sigma(X) \neq X\}$$

This set will always be finite for the substitutions we consider here, so that, in practice, a substitution can be described by the images of the variables X in its domain. Given a type A, we write $A[\sigma]$ for the type A where every variable X has been replaced by $\sigma(X)$. Formally, it is defined by induction on the type A by

$$X[\sigma] = \sigma(X)$$
$$(A \to B)[\sigma] = A[\sigma] \to B[\sigma]$$

We say that a type A is *more general* than a type B, what we write $A \sqsubseteq B$, when there is a substitution σ such that $B = A[\sigma]$.

Lemma 4.4.2.1. The relation $\sqsubseteq$ is a partial order on types modulo α-conversion.

Given a context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, we also write

$$\Gamma[\sigma] = x_1 : A_1[\sigma], \ldots, x_n : A_n[\sigma]$$

In this case, we sometimes say that the context $\Gamma[\sigma]$ is a *refinement* of the context Γ. It is easily shown that if a term admits a type, it also admits a less general type: lemma 4.4.1.1 generalizes as follows.

Lemma 4.4.2.2. Given a term t such that $\Gamma \vdash t : A$ is derivable and a substitution σ then $\Gamma[\sigma] \vdash t : A[\sigma]$ is also derivable.

Proof. By induction on the derivation of $\Gamma \vdash t : A$. □

Definition 4.4.2.3 (Principal type). Given a context Γ and a λ-term t, a *principal type* (or *most general type*) for t in the context Γ consists of a substitution σ and a type A such that

– $\Gamma[\sigma] \vdash t : A$ is derivable,

– for every substitution τ such that $\Gamma[\tau] \vdash t : B$ is derivable, there exists a substitution τ' such that $\tau = \tau' \circ \sigma$ and $B = A[\tau']$.

In other words, the most general type is a type A for t in some refinement of the context Γ such that every other type can be obtained by substitution, in the sense of lemma 4.4.2.2.

This is often used in the case where the context Γ is empty, in which case the substitution σ is not relevant. In this case, the principal type for t is a type A such that $\vdash t : A$ is derivable and which is minimal: given a type B, we have $\vdash t : B$ derivable if and only if $A \sqsubseteq B$.

Example 4.4.2.4. The principal type for $t = \lambda x.x$ is $X \to X$: the types of t are those of the form $A \to A$ for some type A.

Example 4.4.2.5. The principal types for the λ-terms

$$\lambda xyz.(xz)(yz) \qquad \text{and} \qquad \lambda xy.x$$

are respectively

$$(X \to Y \to Z) \to (X \to Y) \to X \to Z \qquad \text{and} \qquad X \to Y \to X$$

4.4.3 Computing the principal type. We now give an algorithm to compute the principal type of a λ-term. A *type equation system* is a finite set

$$E = \{A_1 \mathrel{\overset{?}{=}} B_1, \ldots, A_n \mathrel{\overset{?}{=}} B_n\} \tag{4.2}$$

consisting of pairs types A_i and B_i, written $A_i \mathrel{\overset{?}{=}} B_i$ and called *type constraints*. A substitution σ is a *solution* of the equation system (4.2) when applying it makes every equation of E valid, i.e. for every $1 \leqslant i \leqslant n$ we have

$$A_i[\sigma] = B_i[\sigma]$$

Typing with constraints. The idea is that to every context Γ and λ-term t, we are going to associate a type A and a type equation system E which are *complete* in the sense that

– for every solution σ of E, we have

$$\Gamma[\sigma] \vdash t : A[\sigma]$$

– if there is a substitution σ such that

$$\Gamma[\sigma] \vdash t : B$$

then σ is a solution of E such that $B = A[\sigma]$.

In this sense, the solutions of E describe all the possible types of t in the refinements of the context Γ. Its elements are sometimes called *constraints* since they encode constraints on acceptable substitutions. We will do so by imposing the "minimal amount of equations" to E so that t admits a type A in the context Γ. As usual, this is performed by induction on t, distinguishing the three possible cases:

- x: we have a type A if and only if $x \in \mathrm{dom}(\Gamma)$, in which case $A = \Gamma(x)$,

- $\lambda x.t$: the type A should be of the form $B \to C$ for where C is the type of t. Writing A_t for the type inferred for t, we thus define $A = X \to A_t$ for some fresh variable X,

- $t\,u$: we have a type A if and only if t is of the form $B \to A$ and u is of type B. Writing A_t for the type inferred for t and A_u for the type inferred for u, we thus define $A = X$ for some fresh variable X and add the equation

$$A_t \overset{?}{=} (A_u \to X)$$

Above, the fact that X is "fresh" means that it does not occur somewhere else (in the contexts, the types or the equation systems).

Sequent presentation. More formally, this can be presented in the form of a "sequent" calculus, where the sequents are of the form

$$\Gamma \vdash t : A \mid E$$

where Γ is a context, t is a term, A is a type and E is a type equation system: given Γ and t, a the derivation of such a sequent will be seen as producing the type A and the equations E. The rules are

$$\frac{}{\Gamma \vdash x : \Gamma(x) \mid} \ (\mathrm{ax}) \qquad\qquad \text{with } x \in \mathrm{dom}(\Gamma)$$

$$\frac{\Gamma, x : X \vdash t : A_t \mid E_t}{\Gamma \vdash \lambda x.t : X \to A_t \mid E_t} \ (\to_{\mathrm{I}}) \qquad\qquad \text{with } X \text{ fresh}$$

$$\frac{\Gamma \vdash t : A_t \mid E_t \qquad \Gamma \vdash u : A_u \mid E_u}{\Gamma \vdash t\,u : X \mid E_t \cup E_u \cup \{A_t \overset{?}{=} (A_u \to X)\}} \ (\to_{\mathrm{E}}) \qquad \text{with } X \text{ fresh}$$

Example 4.4.3.1. For instance, for the term $\lambda fx.f\,x$, we have the following derivation:

$$\frac{\dfrac{\dfrac{}{f : Z, x : X \vdash f : Z \mid} \ (\mathrm{ax}) \qquad \dfrac{}{f : Z, x : X \vdash x : X \mid} \ (\mathrm{ax})}{\dfrac{f : Z, x : X \vdash f\,x : Y \mid Z = X \to Y}{\dfrac{f : Z \vdash \lambda x.f\,x : X \to Y \mid Z = X \to Y}{\vdash \lambda f.\lambda x.f\,x : Z \to (X \to Y) \mid Z = X \to Y} \ (\to_{\mathrm{I}})} \ (\to_{\mathrm{I}})}}{} \ (\to_{\mathrm{E}})$$

The type A and the equations E describe exactly all the possible types for t in the context Γ in the following sense.

Lemma 4.4.3.2. Suppose that $\Gamma \vdash t : A \,|\, E$ is derivable using the above rules, then

– for every solution σ of E the sequent $\Gamma[\sigma] \vdash t : A[\sigma]$ is derivable (in the sense of section 4.1.4),

– if there is a substitution σ and a type B such that $\Gamma[\sigma] \vdash t : B$ is derivable then σ is a solution of E and $B = A[\sigma]$.

Proof. By induction on the derivation of $\Gamma \vdash t : A \,|\, E$. □

It is easily seen that given a context Γ and term t there is exactly one type A and system E such that $\Gamma \vdash t : A \,|\, E$ is derivable (up to the choice of type variables), we can thus speak of the type A and the system E associated to a term t in a context Γ. Moreover, the above rules are easily translated into a method for computing those. An implementation of the resulting algorithm is provided in figure 4.5: the function `infer` generates, given a an environment `env` describing the context Γ, the type A and the equation system E, encoded as a list of pairs of terms.

Computing the principal type. What is not clear yet is

– how to compute the solutions of E,

– how to compute the most general type for t in the context Γ.

We will see in section 5.4 that if a system of equations admits a solution then it admits a most general one: provided there is a solution, there is a solution σ such that the solutions are exactly substitutions of the form $\tau \circ \sigma$ for some substitution τ. Moreover, we will see an algorithm to actually compute this most general solution: this is called the *unification* algorithm. This finally provides us with what we were looking for.

Theorem 4.4.3.3. Suppose given a context Γ and a term t. Consider the type A and the system E such that $\Gamma \vdash t : A \,|\, E$ is derivable, and write σ for the most general solution of E. Then the substitution σ together with the type $A[\sigma]$ is a principal type for t in the environment Γ.

In-place unification. In practice, people do not implement the computation of most general types by first generating equations and then solving them, although there are notable exceptions [PR05]: we can directly change the value of the variables instead of deferring this using equations. Moreover, this can be done efficiently by using references. We will see in section 5.4.5 that unification can always be performed this way, and only describe here the implementation specialized to our problem.

Instead of generating equations, we can replace type variables as follows:

– when we have an equation of the form $X \overset{?}{=} A$, we can directly replace X by A, provided that $X \notin \mathrm{FV}(A)$,

– when we have an equation of the form $A \overset{?}{=} X$, we can directly replace X by A, provided that $X \notin \mathrm{FV}(A)$,

```
(** Types *)
type ty =
  | TVar of int
  | TArr of ty * ty

(** Generate a fresh type variable. *)
let fresh =
  let n = ref (-1) in fun () -> incr n; TVar !n

(** Terms. *)
type term =
  | Var of string
  | Abs of string * term
  | App of term * term

(** Type constraints. *)
type teq = (ty * ty) list

(** Type and equations. *)
let rec infer env : term -> ty * teq = function
  | Var x -> List.assoc x env, []
  | Abs (x, t) ->
    let ax = fresh () in
    let at, et = infer ((x,ax)::env) t in
    TArr (ax, at), et
  | App (t, u) ->
    let at, et = infer env t in
    let au, eu = infer env u in
    let ax = fresh () in
    ax, (at, TArr (au, ax))::(et@eu)
```

Figure 4.5: Typability with constraints in OCaml.

– when we have an equation of the form $(A \to B) \overset{?}{=} (A' \to B')$, we can replace it by the two equations $A \overset{?}{=} A'$ and $B \overset{?}{=} B'$, and recursively act on those.

In order to perform this efficiently, we change the representation of type variables to the following:

```
(** Types *)
type ty =
  | TVar of tvar ref
  | TArr of ty * ty

(** Type variables. *)
and tvar =
  | Link of ty  (* a substituted type variable *)
  | AVar of int (* a type variable *)
```

A variable, corresponding to the constructor `TVar`, is now a reference, meaning that its value can be changed. Initially, this reference will point to a value of the form `AVar` n, meaning that it is the variable with number n. However, we can replace its contents by another type A, in which case we make the reference point to a value of the form `Link` A (it is a "link" to the type A): this method has the advantage of changing at once the contents of all the occurrences of the variable. The type `tvar` thus indicates the possible values for a variable: it is either a real variable (`AVar`) or a substituted variable (`Link`). With this representation, a variable containing a link to a type A should be handled as if it was the type A. To this end, we implement a function which will remove links at the top level of types:

```
let rec unlink = function
  | TArr (a, b) -> TArr (a, b)
  | TVar v as a ->
      match !v with
      | Link a -> unlink a
      | AVar _ -> a
```

In order to check the side condition $X \notin \mathrm{FV}(A)$ above, we need to implement a function which checks whether a variable X *occurs* in a type A, i.e. whether $X \in \mathrm{FV}(A)$. This easily done by induction on A:

```
let rec occurs x = function
  | TArr (a, b) -> occurs x a || occurs x b
  | TVar v ->
    match !v with
    | Link a -> occurs x a
    | AVar _ as y -> x = y
```

Next, instead of generating an equation $A \overset{?}{=} B$, we will use the following function which will replace the type variables in A and B following the method described above, called *unification*:

```
let rec unify a b =
  match unlink a, b with
```

```
  | TVar v, b -> assert (not (occurs !v b)); v := Link b
  | a, TVar v -> v := Link a
  | TArr (a, b), TArr (a', b') -> unify a a'; unify b b'
```

Finally, the type inference algorithm can be implemented as before, except that we do not return the equations anymore, only the type, since type variables are changed in place: in the case of application, instead of generating the equation $A_t \stackrel{?}{=} (A_u \to X)$, we instead call the function `unify` which will replace type variables in a minimal way needed to make the types A_t and $A_u \to X$ equal:

```
let rec infer env = function
  | Var x ->
    List.assoc x env
  | App (t, u) ->
    let a = infer env u in
    let b = fresh () in
    unify (infer env t) (TArr (a,b));
    b
  | Abs (x, t) ->
    let a = fresh () in
    let b = infer ((x,a)::env) t in
    TArr (a, b)
```

Example 4.4.3.4. The term $\lambda f x.f\, x$ can be represented as the term

```
Abs ("f", Abs ("x", App (Var "f", Var "x")))
```

If we infer its type (in the empty environment) using the above function `infer`, we obtain the following result

```
Arr
 (TVar
   {contents =
     Link
       (TArr (TVar {contents = AVar 1}, TVar {contents = AVar 2}))
   },
 TArr (TVar {contents = AVar 1}, TVar {contents = AVar 2}))
```

which is OCaml's way of saying

$$(X \to Y) \to (X \to Y)$$

(in OCaml, references are implemented as records with one mutable field labeled `contents`).

Remark 4.4.3.5. In the unification function, when facing an equation $X \stackrel{?}{=} A$, it is important to check that X does not occur in A. For instance, let us try to type $\lambda x.xx$, which is not expected to be typable. The inference will roughly proceed as follows.

1. Since it is an abstraction, the type of $\lambda x.xx$ must be of the form $X \to A$, where A is the type of xx. Let's find the type of xx assuming x of type X.

2. The term xx is an application whose function is x of type X and argument is x of type X. We must therefore have $X \stackrel{?}{=} (X \to Y)$ and the type of xx is Y.

With the above implementation, the algorithm will raise an error: the unification of X and $X \to Y$ will fail because $X \in \mathrm{FV}(X \to Y)$. If we forgot to check this, we would generate for x the type $X \to Y$ where X *is* (physically) the type itself. This would intuitively correspond to allowing the infinite type

$$(((\ldots \to Y) \to Y) \to Y) \to Y$$

which should not be allowed.

Typability. The above algorithms can also be used to decide the *typability* of a term t, i.e. answer the following question: is there a context in which t admits a type?

Theorem 4.4.3.6. The typability problem for λ-calculus is decidable.

Proof. Suppose given a term t. We write $\mathrm{FV}(t) = \{x_1, \ldots, x_n\}$ for the set of free variables and define the context $\Gamma = x_1 : X_1, \ldots, x_n : X_n$. Using lemma 4.1.5.1, it is not difficult to show that t admits a type if and only if it admits a type in the context Γ, which can be decided as above. $\square$

4.4.4 Hindley-Milner type inference. In this section, we go on a small excursion and investigate polymorphic types. We have seen that a Curry-style λ-term usually admits multiple types. However, a given term cannot be used within a same term with two different types. In a real-world programming language this is a problem: for instance, if we define the identity function, we cannot apply it both to integers and strings. If we want to do so, we need to define two identity functions, one for integers and one for strings, with the same definition. One way to overcome this problem is to allow functions to be *polymorphic*, i.e. to have multiple types. For instance, we will be able to type the identity as

$$\forall X.X \to X$$

meaning that it has type $X \to X$ for any possible value of the variable X. OCaml features such types: variables beginning by a ' are implicitly universally quantified, so that the identity has type `'a -> 'a`.

Type schemes. Formally, a *type* A is defined as before, and *type schemes* $\underline{A}$ are generated by the grammar

$$\underline{A} ::= A \mid \forall X.\underline{A}$$

where X is a type variable and A is a type. In other words, a type scheme is a type with some universally quantified variables at top level, i.e. a formula of the form

$$\forall X_1.\ldots \forall X_n.A$$

Having such a "type" for a term means that it can have any type in the set

$$\llbracket \underline{A} \rrbracket = \{A[A_1/X_1, \ldots, A_n/X_n] \mid A_1, \ldots, A_n \text{ types}\}$$

i.e. any type obtained by replacing the universally quantified type variables by some types. As usual, in a type scheme $\forall X.\underline{A}$, the variable X is *bound* in $\underline{A}$ and could be renamed. The *free variables* of a type scheme are

$$\mathrm{FV}(\forall X_1.\ldots \forall X_n.A) = \mathrm{FV}(A) \setminus \{X_1, \ldots, X_n\}$$

Given a variable X and a type B, we write $\underline{A}[B/X]$ for the type scheme $\underline{A}$ where the variable X has been replaced by B (as usual, one has to properly take care of bound variables):

$$(\forall X_1.\ldots\forall X_n.A)[B/X] = \forall X_1.\ldots\forall X_n.A[B/X]$$

whenever $X_i \notin \mathrm{FV}(B)$ whenever $1 \leqslant i \leqslant n$. We consider type schemes modulo α-conversion, which can be defined by:

$$\forall X.\underline{A} = \forall Y.\underline{A}[Y/X]$$

We write $\underline{A} \sqsubseteq \underline{B}$ when the set of types described by $\underline{A}$ is included in the set of types of $\underline{B}$, i.e. $[\![\underline{A}]\!] \subseteq [\![\underline{B}]\!]$. In this case, we say that the type scheme $\underline{A}$ is *more general* than $\underline{B}$, and that $\underline{B}$ is *less general* or a *specialization* of $\underline{A}$.

Lemma 4.4.4.1. We have

$$\forall X_1.\ldots\forall X_n.A \sqsubseteq \forall Y_1.\ldots\forall Y_n.B$$

if and only if there are types $A_1,\ldots,A_n$ such that $B = A[A_1/X_1,\ldots,A_n/X_n]$ and the Y_i are variables which are not free in $\forall X_1.\ldots\forall X_n.A$.

When we have $\underline{A} \sqsubseteq \underline{B}$, which means that $\underline{B}$ was obtained from $\underline{A}$ by replacing some universally quantified variables X_i by types A_i, but not only: we can also universally quantify some of the fresh variables introduced by the A_i afterwards. For instance, we have

$$\forall X.X \to X \sqsubseteq \forall Y.(Y \to Y) \to (Y \to Y) \sqsubseteq (Z \to Z) \to (Z \to Z)$$

Hindley-Milner typing system. We are now going to give a typing system for a programming language whose terms are

$$t, u ::= x \mid \lambda x.t \mid t\,u \mid \mathtt{let}\,x = t\,\mathtt{in}\,u$$

Compared to λ-calculus, the only new construction is $\mathtt{let}\,x = t\,\mathtt{in}\,u$, and means that we should declare x to be t in the term u. From an operational point of view, it is thus the same as $(\lambda x.u)\,t$. The two constructions however differ from the typing point of view: the type of a variable defined by a $\mathtt{let}$ will be *generalized*, which means that we are going to universally quantify the type variables we can, so that the type becomes polymorphic. A variable declared by a $\mathtt{let}$ can thus be used with multiple types, which is not the case for an argument of a function. For instance, in OCaml, we can define the identity once with a $\mathtt{let}$, and use it on an integer and on a string:

```
let () =
  let id = fun x -> x in
  print_int (id 3); print_string (id "a")
```

This is allowed because the type inferred for `id` is $\forall X.X \to X$, which is polymorphic. On the other hand, the following code is rejected:

```
let () =
  (fun id ->
    print_int (id 3); print_string (id "a")
  ) (fun x -> x)
```

Namely, the type inferred for the argument `id` of the function is $X \to X$. During the type inference, OCaml sees that it is applied to `3`, and therefore replaces X by `int`, i.e. it guesses that the type of `id` must be `int` $\to$ `int` and thus raises a type error when we also apply it to a string. The identity argument is monomorphic: it can be applied to an integer, or to a string, but not both.

We now present an algorithm, due to Hindley and Milner [Hin69, Mil78] which infers such types. A *context* Γ is a list

$$x_1 : \underline{A}_1, \ldots, x_n : \underline{A}_n$$

consisting of pairs of variables and type schemes. The free variables of such a context are

$$\mathrm{FV}(\Gamma) = \bigcup_{i=1}^{n} \mathrm{FV}(A_i)$$

We will consider sequents of the form $\Gamma \vdash t : A$ where Γ is a context, t a term and A a type: we still infer a type (as opposed to a type scheme) for a term. The rules for our typing system, which assigns type schemes to terms are the following ones:

$$\frac{\Gamma(x) = \underline{A} \qquad \underline{A} \sqsubseteq B}{\Gamma \vdash x : B} \ (\mathrm{ax}) \qquad\qquad \frac{\Gamma \vdash t : A \qquad \Gamma, x : \forall_\Gamma A \vdash u : B}{\Gamma \vdash \mathtt{let}\, x = t \,\mathtt{in}\, u : B} \ (\mathrm{let})$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B} \ (\to_\mathrm{E}) \qquad\qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \ (\to_\mathrm{I})$$

The rules $(\to_\mathrm{E})$ and $(\to_\mathrm{I})$ for elimination and introduction of functions are the usual ones. The rule (ax) allows to specialize the type of a variable in the context: if x has type $\underline{A}$ in the context Γ, then we can assume that it actually has any type B with $\underline{A} \sqsubseteq B$: with our above example, if `id` has the type scheme $\forall X.X \to X$ in the context, then we can assume that it has type `int` $\to$ `int` (or `string` $\to$ `string`) when we use it, and we can make different assumptions at each use. Finally, the rule `let` states that if we can show that t has type A, then we can assume that it has the more general type scheme

$$\forall_\Gamma A = \forall X_1. \ldots . \forall X_n . A$$

where $\mathrm{FV}(A) \setminus \mathrm{FV}(\Gamma) = \{X_1, \ldots, X_n\}$, called the *generalization* of A with respect to Γ. We thus universally quantify over all type variables which are not already present in the context.

Remark 4.4.4.2. In the rule (let), it is important that $\forall_\Gamma A$ does not universally quantify over all the variables in A, but only those in $\mathrm{FV}(A) \setminus \mathrm{FV}(\Gamma)$. Suppose that we did not have this restriction and quantify over all the variables of A. We would then have the derivation

$$\frac{\dfrac{\overline{x : X \vdash x : X} \ (\mathrm{ax}) \qquad \overline{x : X, y : \forall X.X \vdash y : Y} \ (\mathrm{ax})}{x : X \vdash \mathtt{let}\, y = x \,\mathtt{in}\, y : Y} \ (\mathrm{let})}{\vdash \lambda x.\mathtt{let}\, y = x \,\mathtt{in}\, y : X \to Y} \ (\to_\mathrm{I})$$

This is clearly incorrect since the term $\lambda x.\mathtt{let}\, y = x \,\mathtt{in}\, y$ is essentially the identity, and thus should have $X \to X$ as principal type.

The following proposition shows that this typing system amounts to the simple one of section 4.4.1, where it would allow us to infer the type of an expression each time we use it:

Proposition 4.4.4.3. The sequent $\Gamma \vdash \mathtt{let}\, x = t\, \mathtt{in}\, u : A$ is derivable if and only if t is typable in the context Γ and $\Gamma \vdash u[t/x] : A$ is derivable.

Algorithm W. Formulated as above, it is not clear how to write down an algorithm which would infer the most general type of a term. The problem lies in the (ax) rule: given a variable x which has type scheme $\underline{A}$ in the context, we have to come up with a type B which specializes $\underline{A}$, and there is no obvious way of doing so. Instead, we will replace all the universally quantified variables of $\underline{A}$ by fresh variables, and will gradually compute a substitution which will fill those in. The resulting algorithm is called *algorithm W* [DM82]. It is very close to the algorithm we have seen in section 4.4.3 except that, instead of generating type equations and solving them afterward in order to obtain a substitution, we compute the substitution during the inference. We can express this algorithm using sequents of the form

$$\Gamma \vdash t : A \,|\, \sigma$$

where Γ is a context, t a term, A a type and σ a substitution. The rules are the following ones, they should be read as producing A and σ from Γ and t:

$$\frac{\Gamma(x) = \underline{A}}{\Gamma \vdash x : \,!\underline{A} \,|\, \mathrm{id}} \;(\mathrm{ax})$$

$$\frac{\Gamma, x : X \vdash t : B \,|\, \sigma \qquad X \text{ fresh}}{\Gamma \vdash \lambda x.t : X[\sigma] \to B \,|\, \sigma} \;(\to_{\mathrm{I}})$$

$$\frac{\Gamma \vdash t : C \,|\, \sigma \qquad \Gamma \vdash u : A \,|\, \sigma' \qquad X \text{ fresh} \qquad \sigma'' = \mathrm{mgu}(A \to X, C)}{\Gamma \vdash t\,u : X[\sigma''] \,|\, \sigma'' \circ \sigma' \circ \sigma} \;(\to_{\mathrm{E}})$$

$$\frac{\Gamma \vdash t : A \,|\, \sigma \qquad \Gamma[\sigma], x : \forall_{\Gamma[\sigma]} A \vdash u : B \,|\, \sigma'}{\Gamma \vdash \mathtt{let}\, x = t\, \mathtt{in}\, u : B \,|\, \sigma' \circ \sigma} \;(\mathrm{let})$$

Those can be explained as follows.

(ax) Given the type scheme $\underline{A}$ associated to the variable x in the context, we declare that the type for x is $!\underline{A}$ under the identity substitution. Here, $!\underline{A}$ is a notation for the *instantiation* of $\underline{A}$, by which we mean that we have replaced all the universally quantified variables by fresh ones (i.e. variables which do not already occur in Γ). If the type scheme $\underline{A}$ is $\forall X_1. \ldots \forall X_n.A$, the type $!\underline{A}$ is thus

$$\underline{A}[Y_1/X_1, \ldots, Y_n/X_n]$$

where the variables Y_i are fresh and distinct.

($\to_{\mathrm{I}}$) In order to infer the type of $\lambda x.t$, we have to guess a type for x and infer the type of t in the context where x has this type. Since we have no idea of what this type should be, we simply infer the type of t in the environment where x has type X, a fresh type variable. This will result in a type B

and a substitution σ such that $\Gamma[\sigma], x : X[\sigma] \vdash t : B$ and therefore we can deduce that $\lambda x.t$ has type $X[\sigma] \to B$.

($\to_{\text{E}}$) We first infer a type A for u, and the type C for t. In order for $t\,u$ to be typable C should be of the form $A \to B$. We therefore use the unification procedure described in section 5.4 in order to compute the most general substitution σ'' such that $\sigma''(A \to X) = \sigma(B)$ for some fresh variable X, and we will have $B = X[\sigma]$; this substitution is written $\sigma'' = \text{mgu}(A \to X, C)$ (here, "mgu" means *most general unifier*, see section 5.4.2). We deduce that $t\,u$ has the type B we have computed.

(let) There is no real novelty in this rule compared to earlier. In order to infer the type of $\text{let}\, x = t \,\text{in}\, u$, we infer a type A for t and then infer a type B for u in the environment where x has the type scheme obtained by generalizing A with respect to Γ.

This algorithm generates a valid type according to the previous rules:

Theorem 4.4.4.4 (Correctness). If $\Gamma \vdash t : A \,|\, \sigma$ is derivable then $\Gamma[\sigma] \vdash t : A$ is derivable.

Moreover, it is actually the most general one that could be inferred:

Theorem 4.4.4.5 (Principal types). Suppose that $\Gamma \vdash t : A \,|\, \sigma$ is derivable. Then, for every substitution τ and type B such that $\Gamma[\tau] \vdash t : B$ there exists a substitution τ' such that $\tau = \tau' \circ \sigma$ and $B = A[\tau']$.

Example 4.4.4.6. Here are some principal types which can be computed with the algorithm:

$$\lambda x.\text{let}\, y = x \,\text{in}\, y : X \to X$$
$$\lambda x.\text{let}\, y = \lambda z.x \,\text{in}\, y : X \to Y \to X$$
$$\lambda x.\text{let}\, y = \lambda z.x\, z \,\text{in}\, y : (X \to Y) \to (X \to Y)$$

Implementing algorithm W. Algorithm W can be coded by suitably implementing the above rules. We define the type of terms as

```
type term =
  | Var of var
  | App of term * term
  | Abs of var  * term
  | Let of var  * term * term
```

where `var` is an alias for `int` for clarity. Then type schemes are encoded as the following type:

```
type ty =
  | EVar of int     (* non-quantified variable *)
  | UVar of int     (* universally quantified variable *)
  | TArr of ty * ty
```

Here, instead of universally quantifying some variables, we use two constructors: `UVar` n is a variable which is universally quantified, and `EVar` n is a variable which is not. The generation of fresh type variables can be achieved with a counter, as usual:

```
let fresh =
  let n = ref (-1) in fun () -> incr n; EVar !n
```

Next, the instantiation of a type scheme is performed by replacing each universally quantified variable with a fresh, non-quantified, one (we use a list `tenv` in order to remember when a universal variable has already been replaced by some variable, in order to always replace it by the same variable):

```
let inst =
  let tenv = ref [] in
  let rec inst = function
    | UVar x ->
      if not (List.mem_assoc x !tenv) then
        tenv := (x, fresh ()) :: !tenv;
      List.assoc x !tenv
    | EVar x -> EVar x
    | TArr (a, b) -> TArr (inst a, inst b)
  in
  inst
```

The following function checks whether a variable occurs in a type:

```
let rec occurs x = function
  | EVar y      -> x = y
  | UVar _      -> false
  | TArr (a, b) -> occurs x a || occurs x b
```

We can then generalize a type with respect to a given context by changing each variable EVar n which does not occur in the context into the corresponding universally quantified variable UVar n:

```
let rec gen env = function
  | EVar x ->
    if List.exists (fun (_,a) -> occurs x a) env
    then EVar x else UVar x
  | UVar x -> UVar x
  | TArr (a, b) -> TArr (gen env a, gen env b)
```

We can finally implement the function which will infer the type of a term in a given environment and return it together with the corresponding substitution. The four cases of the match correspond to the four different rules above:

```
let rec infer env = function
  | Var x ->
    let a =
      try List.assoc x env
      with Not_found -> raise Type_error
    in
    inst a, Subst.id
  | Abs (x, t) ->
    let a = fresh () in
    let b, s = infer ((x,a)::env) t in
    TArr (Subst.app s a, b), s
```

```
  | App (t, u) ->
    let a, su = infer env u in
    let b = fresh () in
    let c, st = infer env t in
    let s = unify (TArr (a, b)) c in
    Subst.app s b, Subst.comp s (Subst.comp su st)
  | Let (x, t, u) ->
    let a, st = infer env t in
    let b, su = infer ((x,gen (Subst.app_env st env) a)::env) u in
    b, Subst.comp su st
```

We have implemented substitutions as functions `int -> ty` associating a type to a type variable. The functions `Subst.id`, `Subst.comp`, `Subst.app` and `Subst.app_env` respectively compute the identity substitution, the composite of substitutions and the application of a substitution to a term and to an environment. Their implementation is left to the reader. Finally, above, the function `unify` implements the unification algorithm described in section 5.4:

```
let rec unify l =
  match l with
  | (EVar x, b)::l ->
    if occurs x b then raise Type_error;
    Subst.comp (unify l) (Subst.make [x, b])
  | (a, EVar x)::l ->
    unify ((EVar x, a)::l)
  | (TArr (a, b), TArr (a', b'))::l ->
    unify ([a, a'; b, b']@l)
  | (UVar _, _)::_ | (_, UVar _)::_ -> assert false
  | [] -> Subst.id
let unify a b = unify [a, b]
```

Algorithm J. The previous algorithm is theoretically nice. In particular, it is well adapted to making correctness proofs. However, it is quite inefficient: we have to apply substitutions to many types (including to the context) and we have to go through the context to look for type variables which have been used. As in section 4.4.3, the solution is to modify type variables in-place by using references. The resulting algorithm is sometimes called *algorithm J.* We thus change the implementation of types to

```
type ty =
  | EVar of tvar ref (* non-quantified variable *)
  | UVar of int      (* universally quantified variable *)
  | TArr of ty * ty
and tvar =
  | Unbd of int (* unbound variable *)
  | Link of ty  (* substituted variable *)
```

Most functions are adapted straightforwardly. The main novelty is in the unification function, which now performs the modification of types in-place:

```
let rec unify a b =
  match unlink a, unlink b with
```

```
  | EVar x, _ ->
    if occurs x b then raise Type_error else x := Link b
  | _, EVar y -> unify b a
  | TArr (a1, a2), TArr (b1, b2) -> unify a1 b1; unify a2 b2
  | _ -> raise Type_error
```

and the type inference function which is simpler to write, because it does not need to propagate the substitutions:

```
let rec infer env = function
  | Var x -> (try inst (List.assoc x env)
              with Not_found -> raise Type_error)
  | Abs (x, t) ->
    let a = fresh () in
    let b = infer ((x,a)::env) t in
    TArr (a, b)
  | App (t, u) ->
    let a = infer env u in
    let b = fresh () in
    let c = infer env t in
    unify (TArr (a,b)) c;
    b
  | Let (x, t, u) ->
    let a = infer env t in
    infer ((x, gen env a)::env) u
```

The substitutions are now performed very efficiently because we do not have to go through terms anymore: references are doing the job for us. There is, however, one last source of inefficiency in this code: in the function `unify`, the function `occurs x b` has to go through all the type `b` to see whether the variable `x` occurs in it or not. There is a very elegant solution to this due to Rémy [Rém92] that we learned from [Kis13]. To each type variable, we are going to assign an integer called its *level*, which indicates the depth of let-declaration when it was created. Initially, the level is 0 by convention and in an expression $\text{let } x = t \text{ in } u$ at level n, the variables created by t will have level $n+1$, whereas the variables of u will still have level n (it is some sort of de Bruijn index). For instance, in the term

$$\text{let } a = (\text{let } b = \lambda x.x \text{ in } \lambda y.y) \text{ in } \lambda z.z$$

the type variables associated to x, y and z will have level 2, 1 and 0 respectively. This can be figured graphically as follows:

level 2:	$\lambda x.x$
level 1:	$(\text{let } b = \quad\quad \text{ in } \lambda y.y)$
level 0:	$\text{let } a = \quad\quad\quad\quad\quad\quad \text{ in } \lambda z.z$

One can convince himself that, in a term $\text{let } x = t \text{ in } u$, the variables which should be generalized in the rule (let) are those which were "locally created" during the inference of t, i.e. those which are at a strictly higher level than the current one. We thus modify our implementation once more. We begin by

declaring a global reference, which will record the current level when performing the type inference, along with two functions in order to increase and decrease the current level:

```
let level = ref 0
let enter_level () = incr level
let leave_level () = decr level
```

We also change the representation of type variables: the constructor for unbound variables becomes

```
  | Unbd of int * int (* unbound variable (name / level) *)
```

It now takes two integers as argument: the number of the variable (acting as its name) and its level. In the generalization function, we only generalize variables which are above the current level:

```
let rec gen a =
  match a with
  | EVar x ->
    if tlevel x <= !level then EVar x
    else UVar (tname x)
  | UVar x -> UVar x
  | TArr (a, b) -> TArr (gen a, gen b)
```

where `tname` and `tlevel` respectively return the name and type of a type variable. Finally, levels get updated in the `infer` function whose only change is in the `Let` case:

```
  | Let (x, t, u) ->
    enter_level ();
    let a = infer env t in
    leave_level ();
    infer ((x, gen a)::env) u
```

We increase the current level when typechecking the definition and decrease it afterward.

Example 4.4.4.7. In the function

$$\lambda x.\mathsf{let}\, y = \lambda z.z \,\mathsf{in}\, y$$

the type variable Z associated to z has level 1, so that it gets generalized in the type of y, because y is declared at level 0 and $0 < 1$: in the environment, y will have the type scheme $\forall Z.Z \to Z$. However, in the function

$$\lambda x.\mathsf{let}\, y = \lambda z.x \,\mathsf{in}\, y$$

the type variable X associated to x does not get generalized because it is of level 0, so that y has the type scheme $\forall Y.Y \to X$, and not $\forall X.\forall Y.Y \to X$.

There is a catch however: it might happen that, during unification (see the function `unify` above), a variable X with low level ℓ gets substituted with a type A containing variables of high level. In this case, for every variable in A, the level should be lowered to the minimum of this level and ℓ before performing

the substitution: the level gets "contaminated" by the one of the variable it is unified with. However, we are smart and see that the function `occurs` is already going through the type just before we substitute, and it is the only place where it is used, so that we can use it to both check the occurrence and update the levels. We therefore change it to

```
let rec occurs x a =
  match unlink a with
  | EVar y when x = y -> raise Type_error
  | EVar y ->
    let l = tlevel y in
    let l = match !x with Unbd (_,l') -> min l l' | _ -> l in
    y := Unbd (tname y, l)
  | UVar _ -> ()
  | TArr (a, b) -> occurs x a; occurs x b
```

which changes the level of all variables to the minimum of their old level and the current level. Without this modification of `occurs`, for the term

$$\lambda x.\texttt{let } y = \lambda z.x\, z \texttt{ in } y$$

we would infer the unsound type $(X \to Y) \to (Z \to W)$ instead of the expected type $(X \to Y) \to (X \to Y)$.

4.4.5 Bidirectional type checking. We present here another approach to type checking, which does not try to come up with new or most general types: this means that we will fail to infer the type for terms when we are not certain about this type (for instance, if the term can have multiple types). However, we will try to exploit as much as possible the already known type information about terms. This is less powerful than previous methods in the context of λ-calculus, but has the advantage of being simple to implement and of generalizing well to richer logics, where principal types do not exist or type inference is undecidable, see chapter 8.

When implementing type inference, we can see that two different phases are actually involved:

– *type inference*: we come up with a type for the term,

– *type checking*: we make sure that the term has a given type.

For instance, when performing the type inference for a term $t\,u$, we first infer the type for t, which should be of the form $A \to B$, and then we *check* that u has type A. Of course, this checking part is usually done by inferring a type for u and comparing it with A, but in some situations we can exploit the fact that we are checking that the term t has type A, and that this A does bring us some information. For instance, we can check that the term $\lambda x.x$ has the type $X \to X$, but we cannot unambiguously infer a type for $\lambda x.x$ because it admits multiple types (we have seen that there are canonical choices such as the principal type, see section 4.4.2, but here we do not want to make any choice for the user).

This suggests splitting the usual typing judgment $\Gamma \vdash t : A$ in two:

– $\Gamma \vdash t \Rightarrow A$: we *infer* the type A for the term t in the context Γ,

– $\Gamma \vdash t \Leftarrow A$: we *check* that the term t has type A in the context Γ.

We will consider terms of the form

$$t, u ::= x \mid \lambda x.t \mid t\,u \mid (t : A)$$

The only new construction is the last one, $(t : A)$, which means "check that t has type A". It will become handy since it allows bringing type information in terms and is already present in languages such as OCaml, where we can define the identity function on integers by

```
let id = fun x -> (x : int)
```

The rules for type inference and checking are the following ones:

$$\frac{}{\Gamma \vdash x \Rightarrow \Gamma(x)}(\text{ax})$$

$$\frac{\Gamma \vdash t \Rightarrow A \to B \qquad \Gamma \vdash u \Leftarrow A}{\Gamma \vdash t\,u \Rightarrow B}(\to_{\text{E}}) \qquad \frac{\Gamma, x : A \vdash t \Leftarrow B}{\Gamma \vdash \lambda x.t \Leftarrow A \to B}(\to_{\text{I}})$$

$$\frac{\Gamma \vdash t \Leftarrow A}{\Gamma \vdash (t : A) \Rightarrow A}(\text{cast}) \qquad \frac{\Gamma \vdash t \Rightarrow A}{\Gamma \vdash t \Leftarrow A}(\text{sub})$$

They read as follows:

(ax) If we know that x has type A then we can come up with a type for x: namely A.

$(\to_{\text{E}})$ If we can infer a type $A \to B$ for t and check that u has type A then we can infer the type B for $t\,u$.

$(\to_{\text{I}})$ In order to check that $\lambda x.t$ has type $A \to B$, we should check that t has type B when x has type A.

(cast) We can infer the type A for $(t : A)$ provided that t actually has type A.

(sub) This *subsumption* rule states that, as last resort, if we do not know how to check that a term t has type A, we can go back to the old method of inferring a type for it and ensuring that this type is A.

Note that there is no rule for inferring the type of $\lambda x.t$, because there is no way to come up with a type for x without type annotations. Again, this means that we cannot infer a type for the identity $\lambda x.x$, but we can in presence of type annotations:

$$\frac{\dfrac{\dfrac{}{x : A \vdash x \Rightarrow A}(\text{ax})}{\dfrac{x : A \vdash x \Leftarrow A}{\dfrac{\vdash \lambda x.x \Leftarrow A \to A}{\vdash (\lambda x.x : A \to A) \Rightarrow A \to A}(\text{cast})}(\to_{\text{I}})}(\text{sub})}{}$$

An implementation is provided in figure 4.6: the two modes (type inference and checking) are implemented by two mutually recursive functions (`infer` and `check`). There are two kind of errors that can be raised: `Type_error` means that the term is ill-typed as usual, and `Cannot_infer` means that the algorithm could not come up with a type, but the term might still be typable.

Example 4.4.5.1. For illustration purposes, we suppose we have access to real (or float) numbers, of type $\mathbb{R}$, with the rule

$$\overline{\Gamma \vdash r \Rightarrow \mathbb{R}}$$

for every real number r. We also suppose that we have access to the usual mathematical functions (addition, multiplication), as well as a function which computes the mean of a function between two points, i.e. Γ contains

$$\text{mean} : (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} \to \mathbb{R}$$

We can then type

$$\cfrac{\Gamma \vdash \text{mean} \Rightarrow (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} \to \mathbb{R} \qquad \cfrac{\cfrac{\overline{\Gamma, x:\mathbb{R} \vdash x \Rightarrow \mathbb{R}}}{\Gamma, x:\mathbb{R} \vdash x \Leftarrow \mathbb{R}}}{\Gamma \vdash \lambda x.x \Leftarrow \mathbb{R} \to \mathbb{R}} \qquad \cfrac{\Gamma \vdash 5 \Rightarrow \mathbb{R}}{\Gamma \vdash 5 \Leftarrow \mathbb{R}} \qquad \cfrac{\Gamma \vdash 7 \Rightarrow \mathbb{R}}{\Gamma \vdash 7 \Leftarrow \mathbb{R}}}{\Gamma \vdash \text{mean}\,(\lambda x.x)\,5\,7 \Rightarrow \mathbb{R}}$$

However, we cannot infer the type for the function

$$\lambda fxy.(f\,x + f\,y)/2$$

which would be the definition of mean. When defining a function, we have to give its type and cast it accordingly: we can type

$$(\lambda fxy.(f\,x + f\,y)/2 : (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} \to \mathbb{R})$$

This is why in a programming language such as Agda you have to declare the type of a function when defining it:

```
mean : (R → R) → R → R → R
mean f x y = (x + y) / 2
```

Remark 4.4.5.2. If we omit the rule (cast), it is interesting to note that the terms v such that $\Gamma \vdash v \Leftarrow A$ and the terms n such that $\Gamma \vdash n \Rightarrow A$ is derivable for some context Γ and type A are respectively generated by the grammars

$$v ::= \lambda x.t \mid n \qquad\qquad n ::= x \mid n\,v$$

which is precisely the traditional definition of values (also called normal forms) and neutral terms (already encountered in section 3.5.2 for instance).

4.5 Hilbert calculus and combinators

We have seen in section 3.6.3 that every λ-term can be expressed using application and the two combinators S and K, respectively corresponding to the λ-terms

$$\mathsf{S} = \lambda xyz.(xz)(yz) \qquad\qquad \mathsf{K} = \lambda xy.x$$

```
(** Types. *)
type ty =
  | TVar of string
  | TArr of ty * ty

type var = string

(** Terms. *)
type term =
  | Var  of var
  | App  of term * term
  | Abs  of var  * term
  | Cast of term * ty

exception Cannot_infer
exception Type_error

(** Type inference. *)
let rec infer env = function
  | Var x ->
    (try List.assoc x env with Not_found -> raise Type_error)
  | App (t, u) ->
    (
      match infer env t with
      | TArr (a, b) -> check env u a; b
      | _ -> raise Type_error
    )
  | Abs (x, t) -> raise Cannot_infer
  | Cast (t, a) -> check env t a; a

(** Type checking. *)
and check env t a =
  match t , a with
  | Abs (x, t) , TArr (a, b) -> check ((x, a)::env) t b
  | _ -> if infer env t <> a then raise Type_error
```

Figure 4.6: Bidirectional type checking.

We have also seen in example 4.4.2.5 that the principal types of those terms are respectively

$$(X \to Y \to Z) \to (X \to Y) \to X \to Z \qquad \text{and} \qquad X \to Y \to X$$

which means that they respectively have the type

$$(A \to B \to C) \to (A \to B) \to A \to C \qquad \text{and} \qquad A \to B \to A$$

for every types A, B and C.

It is thus natural to consider a typed version of combinatory terms (see section 3.6.3) expressed by rules, where types and contexts are defined as above, and sequents are of the form

$$\Gamma \vdash t : A$$

where Γ is a context, t is a combinatory term and A is a type. The rules are

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \ (\text{ax})$$

$$\frac{}{\Gamma \vdash \mathsf{S} : (A \to B \to C) \to (A \to B) \to A \to C} \ (\mathsf{S})$$

$$\frac{}{\Gamma \vdash \mathsf{K} : A \to B \to A} \ (\mathsf{K})$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B} \ (\to_{\mathrm{E}})$$

where in (ax) we suppose that $x \in \operatorname{dom}(\Gamma)$. If we apply an analogue of the term erasing procedure (section 4.1.7), we obtain the following logical system:

$$\frac{}{\Gamma, A, \Gamma' \vdash A} \ (\text{ax})$$

$$\frac{}{\Gamma \vdash (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C} \ (\mathrm{S})$$

$$\frac{}{\Gamma \vdash A \Rightarrow B \Rightarrow A} \ (\mathrm{K})$$

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B} \ (\Rightarrow_{\mathrm{E}})$$

which is precisely the Hilbert calculus described in section 2.7! In other words, in the same way that natural deduction corresponds, via the Curry-Howard correspondence, to simply typed λ-calculus, Hilbert calculus corresponds to typed combinatory terms. This was first observed by Curry [CF58].

Example 4.5.0.1. We have seen in example 3.6.3.1 that, in combinatory logic, identity could be expressed as

$$\mathsf{I} = \mathsf{S}\,\mathsf{K}\,\mathsf{K}$$

Its typing derivation is

$$\frac{\dfrac{\vdash \mathsf{S} : (A \to (B \to A) \to A) \to (A \to B \to A) \to A \to A}{} \ (\mathsf{S}) \qquad \dfrac{\vdash \mathsf{K} : A \to (B \to A) \to A}{} \ (\mathsf{K})}{\dfrac{\vdash \mathsf{S}\,\mathsf{K} : (A \to B \to A) \to A \to A}{\vdash \mathsf{S}\,\mathsf{K}\,\mathsf{K} : A \to A} \ (\to_{\mathrm{E}}) \qquad \dfrac{\vdash \mathsf{K} : A \to B \to A}{} \ (\mathsf{K}) \quad (\to_{\mathrm{E}})}$$

from which, by term erasure, we recover the proof of $A \Rightarrow A$ in Hilbert calculus given in example 2.7.1.1.

4.6 Classical logic

Since classical logic is an extension of intuitionistic logic, in the sense that we have more rules, we can expect that the Curry-Howard correspondence can be extended to classical logic. For various reasons, it has been thought for a long time that classical logic had no computational contents, one being that naively imposing $A \Leftrightarrow \neg\neg A$ makes all proofs of a given type equal, see section 2.5.4. It was thus somewhat of a surprise when Parigot introduced the $\lambda\mu$-calculus [Par92], which is an extension of the λ-calculus suitable for classical logic. In section 2.5.9, we have analyzed the proof of $\neg A \lor A$ (or rather its encoding in intuitionistic logic). The main ingredient is the ability to "roll back" to a previous proof goal at any point in the proof: we prove $\neg A$ and at some point we change our mind, go back to proving $\neg A \lor A$, and chose proving A instead. In $\lambda\mu$-calculus, this is achieved by a sort of "exception" mechanism: instead of going on with the computation, we might raise an exception which is going to be caught and change the computation flow. However, in this calculus, the exceptions follow a very particular discipline, making them behave not exactly as in usual languages such as OCaml.

4.6.1 Felleisen's $\mathcal{C}$. Let us try to naively try to extend the Curry-Howard correspondence to classical logic. For clarity, we write here $\bot$ instead of 0 for the type corresponding to falsity. Starting from implicative intuitionistic natural deduction, classical logic can be obtained by adding the rule

$$\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A} \; (\neg\neg_{\mathrm{E}})$$

corresponding to double negation elimination. This suggests that we should add a corresponding construction, say $\mathcal{C}(t)$, to our term calculus together with the typing rule

$$\frac{\Gamma \vdash t : \neg\neg A}{\Gamma \vdash \mathcal{C}(t) : A} \; (\neg\neg_{\mathrm{E}})$$

This calculus allows for "static" Curry-Howard correspondence, in a sense similar to theorem 4.1.7.1: there is a bijective correspondence between typing derivation of λ-terms with $\mathcal{C}$ and proofs in natural deduction with double negation elimination.

In order to hopefully extend this to a dynamical correspondence, we need to introduce a notion of reduction, which should correspond to cut elimination. First note that, unlike previously, here we do not need to add an introduction rule for double negation. Instead, recalling that $\neg A = A \to \bot$, we can construct a proof of $\neg\neg A$ from a proof of A as follows:

$$\frac{\dfrac{\dfrac{}{\Gamma, k : A \to \bot \vdash k : A \to \bot} \; (\mathrm{ax}) \qquad \dfrac{\vdots}{\dfrac{\Gamma \vdash t : A}{\Gamma, k : A \to \bot \vdash t : A} \; (\mathrm{wk})}}{\dfrac{\Gamma, k : A \to \bot \vdash k\,t : \bot}{\Gamma \vdash \lambda k^{A \to \bot}.k\,t : (A \to \bot) \to \bot} \; (\to_{\mathrm{I}})} \; (\to_{\mathrm{E}})}$$

In other words, the introduction rule for double negation should be

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \lambda k^{\neg A}.k\, t : \neg\neg A} \; (\neg\neg_{\mathrm{I}})$$

We can therefore "guess" the reduction rule associated to $\mathcal{C}$ by observing the corresponding cut elimination:

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash t : A}}{\Gamma \vdash \lambda k^{\neg A}.k\, t : \neg\neg A} \; (\neg\neg_{\mathrm{I}})}{\Gamma \vdash \mathcal{C}(\lambda k^{\neg A}.k\, t) : A} \; (\neg\neg_{\mathrm{E}}) \quad \rightsquigarrow \quad \cfrac{\pi}{\Gamma \vdash t : A}$$

The β-reduction rule should thus be

$$\mathcal{C}(\lambda k^{\neg A}.k\, t) \longrightarrow_\beta t$$

Note that this rule only makes sense when k does not occur in t, otherwise the bound variable k could escape its scope... This is indeed the main reduction rule associated to $\mathcal{C}$, but it turns out that two more reduction rules are required for $\mathcal{C}$:

$$\mathcal{C}(\lambda k^{\neg A}.k\, t) \longrightarrow_\beta t \qquad\qquad\qquad \text{if } k \notin \mathrm{FV}(t)$$
$$\mathcal{C}(\lambda k^{\neg(A \to B)}.t)\, u \longrightarrow_\beta \mathcal{C}(\lambda k'^{\neg B}.t[\lambda f^{A \to B}.k\,(f\,u)/k])$$
$$\mathcal{C}(\lambda k^{\neg A}.k\,\mathcal{C}(\lambda k'^{\neg A}.t)) \longrightarrow_\beta \mathcal{C}(\lambda k''^{\neg A}.t[k''/k, k''/k'])$$

The second rule states that the application to the argument u goes through under $\mathcal{C}$: if our calculus had products or coproducts, similar rules would have to be added in order to enforce their compatibility with $\mathcal{C}$. The third rule states that we can merge two uses of $\mathcal{C}$ on the same type.

Let us try to understand what this could mean. Suppose given a term v of type $\neg\neg A$. Since $\neg\neg A = (A \to \bot) \to \bot$, this means that v must be an abstraction taking an argument k of type $A \to \bot$ and return a value of type $\bot$, i.e. v will reduce to a term of the form $\lambda k^{A \to \bot}.u$. Since there is no introduction rule for $\bot$ (there is no way of directly constructing a term of type $\bot$), at some point during the evaluation of u, it must apply k to some argument t of type A in order to produce the value of type $\bot$, i.e. v will reduce to $\lambda k^{A \to \bot}.k\, t$. Thus, $\mathcal{C}(v)$ will reduce to $\mathcal{C}(\lambda k^{A \to \bot}.k\, t)$, which will reduce to t. Reduction path is thus

$$\mathcal{C}(v) \xrightarrow{\;*\;}_\beta \mathcal{C}(\lambda k^{\neg A}.u) \xrightarrow{\;*\;}_\beta \mathcal{C}(\lambda k^{\neg A}.k\, t) \longrightarrow_\beta t$$

This means that $\mathcal{C}(v)$ waits for v to apply its argument k to some term t of type A and returns this argument t. The term k can thus be thought of as an analogue of `return` in some languages such as C, or maybe also as the `raise` operator of OCaml which raises exceptions (more on this later on). However, things are more subtle here because the returned term might itself use some of the terms computed during the evaluation of v. In order to see that in action,

let us compute the term associated to the usual proof of $\neg A \vee A$ (see page 63):

$$
\cfrac{
\cfrac{
\cfrac{
\cfrac{
\cfrac{
\cfrac{
\cfrac{
\cfrac{}{k : \neg(\neg A \vee A), a : A \vdash a : A} \text{ (ax)}
}{k : \neg(\neg A \vee A), a : A \vdash \iota_{\mathrm{r}}(a) : \neg A \vee A} (\vee_{\mathrm{I}}^{\mathrm{r}})
}{k : \neg(\neg A \vee A), a : A \vdash k\,\iota_{\mathrm{r}}(a) : \bot} (\neg_{\mathrm{E}})
}{k : \neg(\neg A \vee A) \vdash \lambda a^A . k\,\iota_{\mathrm{r}}(a) : \neg A} (\neg_{\mathrm{I}})
}{k : \neg(\neg A \vee A) \vdash \iota_{\mathrm{l}}(\lambda a^A . k\,\iota_{\mathrm{r}}(a)) : \neg A \vee A} (\vee_{\mathrm{I}}^{\mathrm{l}})
}{k : \neg(\neg A \vee A) \vdash k\,\iota_{\mathrm{l}}(\lambda a^A . k\,\iota_{\mathrm{r}}(a)) : \bot} (\neg_{\mathrm{E}})
}{\vdash \lambda k^{\neg A} . k\,\iota_{\mathrm{l}}(\lambda a^A . k\,\iota_{\mathrm{r}}(a)) : \neg\neg(\neg A \vee A)} (\neg_{\mathrm{I}})
}{\vdash \mathcal{C}(\lambda k^{\neg A} . k\,\iota_{\mathrm{l}}(\lambda a^A . k\,\iota_{\mathrm{r}}(a))) : \neg A \vee A} (\neg\neg_{\mathrm{E}})
$$

As indicated above, the term $\mathcal{C}(\lambda k^{\neg A} . k\,\iota_{\mathrm{l}}(\lambda a^A . k\,\iota_{\mathrm{r}}(a)))$ cannot reasonably reduce to

$$t = \iota_{\mathrm{l}}(\lambda a^A . k\,\iota_{\mathrm{r}}(a))$$

because the variable k occurs in t. The additional rules make it so that it however acts as t, i.e. it states that it is a proof of $\neg A = A \to \bot$, albeit being surrounded by $\mathcal{C}(\lambda k^{\neg A} . k \ldots)$. If, at some point, we use this proof and apply it to some argument u of type A, the term will thus reduce to

$$\mathcal{C}(\lambda k^{\neg A} . k\,\iota_{\mathrm{r}}(u))$$

which in turn will reduce to $\iota_{\mathrm{r}}(u)$ by the reduction rule associated to $\mathcal{C}$. It thus fakes being a proof of $\neg A$ until we actually use this proof and apply it to some argument u of type A, at which point it changes its mind and declares that it was actually a proof of A, namely u. This is exactly the behavior we were describing in section 2.5.2, when explaining that classical logic allows to "resetting proofs".

Variants of the calculus. The operator $\mathcal{C}$ is due to Felleisen [FH92] and the observation that it could be typed by double negation elimination was first made by Griffin [Gri89], see also [SU06, chapter 7]. Many small variations of the calculus are possible. First note that we could add $\mathcal{C}$ (as opposed to $\mathcal{C}(t)$) as a constant to the language, which corresponds to adding double negation elimination as an axiom instead of a rule:

$$\cfrac{}{\Gamma \vdash \mathcal{C} : \neg\neg A \to A}$$

If we instead use Clavius' law instead of double negation, see theorem 2.5.1.1, then we would have defined an operator cc called `callcc` (for *call with current continuation*):

$$\cfrac{}{\Gamma \vdash \mathrm{cc} : (\neg A \to A) \to A}$$

This operator is implemented in languages such as Scheme and $\mathcal{C}$ is a generalization of it: we have $\mathrm{cc}(\lambda k.t) = \mathcal{C}(\lambda k.k\,t)$. Finally, double negation elimination can also be implemented by the rule

$$\cfrac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}$$

which suggests the following variant of the calculus

$$\frac{\Gamma, x : \neg A \vdash t : \bot}{\Gamma \vdash \mu x^A.t : A}$$

This means that we now add a construction $\mu x^A.t$ to our terms which corresponds to

$$\mu x^A.t = \mathcal{C}(\lambda x^{\neg A}.t)$$

in the previous calculus. In the next section, we will see an alternative calculus based on similar ideas, though with nicer and more intuitive reduction rules.

4.6.2 The $\lambda\mu$-calculus. Let us now introduce the $\lambda\mu$-*calculus* [Par92]. We suppose fixed two sorts of variables: the *term variables x, y*, etc. which behave as usual and the *control variables α, β*, etc. which can be thought of as variables of negated types. The terms are generated by the grammar

$$t, u ::= x \mid t\,u \mid \lambda x.t \mid \mu\alpha.t \mid [\alpha]t$$

The first constructions are the usual ones from the λ-calculus. A term of the form $\mu\alpha.t$ should be thought of as a term *catching* an exception named α and a term $[\alpha]t$ as *raising* the exception α with the value t. The reduction will make it so that the place where it is caught is replaced by t. For instance, we will have a reduction

$$t\,(\mu\alpha.u\,([\alpha]v)) \xrightarrow{*} t\,v$$

meaning that during the evaluation of the argument of t, the exception α will be raised with value v and will thus replace the term at the corresponding $\mu\alpha$. The constructor μ is a binder and terms are considered modulo α-equivalence: $\mu\alpha.t = \mu\beta.(t[\beta/\alpha])$. Beware of the unfortunate similarity in notation between raising and substitution.

The three reduction rules of the calculus are

– the usual β-reduction:

$$(\lambda x.t)u \longrightarrow_\beta t[u/x]$$

– the following rule commuting applications and μ-abstractions:

$$(\mu\alpha.t)\,u \longrightarrow_\beta \mu\beta.t[[\beta]-u/[\alpha]-]$$

where the weird notation $[\beta]-u/[\alpha]-$ in the substitution means that we should replace every subterm of t of the form $[\alpha]v$ by $[\beta]vu$,

– the following reduction rule for μ, stating that if we catch exceptions raised on α and immediately re-raise on β, we might as well raise them directly on β:

$$[\beta](\mu\alpha.t) \longrightarrow_\beta t[\beta/\alpha]$$

Additionally, we require the following η-reduction rule, stating that if we catch on α and immediately re-raise on α, we might as well do nothing:

$$\mu\alpha.[\alpha]t \longrightarrow_\eta t$$

when $\alpha \notin \mathrm{FV}(t)$. It is proved in [Par92] that

Theorem 4.6.2.1. The $\lambda\mu$-calculus is confluent.

Remark 4.6.2.2. The translation between previous calculus based on $\mathcal{C}$ and $\lambda\mu$-calculus was already hinted at at the end of previous section: $\mu\alpha.t$ corresponds to $\mathcal{C}(\lambda\alpha.t)$ and $[\alpha]$ corresponds to applying the argument given by $\mathcal{C}$. More formally, the operators cc and $\mathcal{C}$ can be encoded in the $\lambda\mu$-calculus as

$$\text{cc} = \lambda y.\mu\alpha.[\alpha](y\,(\lambda x.\mu\beta.[\gamma]x))$$
$$\mathcal{C} = \lambda y.\mu\alpha.[\beta](y\,(\lambda x.\mu\gamma.[\alpha]x))$$

The intuition is thus that $\mu\alpha.t$ corresponds to some sort of OCaml construction creating an exception and catching it:

```
let exception Alpha of 'a in
try t with Alpha u -> u
```

and $[\alpha]u$ would correspond to raising the exception:

```
raise (Alpha u)
```

However, there are differences. First, the name of the exception is generated on the fly instead of being hard-coded: we have an α-conversion rule for μ binders. More importantly, the exceptions can never escape their scope in $\lambda\mu$-calculus, unlike in OCaml. For instance, consider the following program in OCaml:

```
let f : int -> int =
  let exception Alpha of (int -> int) in
  try fun n -> raise (Alpha (fun x -> n * x))
  with Alpha g -> g

let () = print_int (f 3)
```

Although the exception `Alpha` seems to be caught (the `raise` is surrounded by a `try` / `catch`), executing the program results in

```
Fatal error: exception Alpha(_)
```

meaning that it was not the case: when executing `f 3`, `f` is replaced by its value and the reduction raises the exception. The analogue of this program in $\lambda\mu$ is

$$f = \mu\alpha.[\alpha](\lambda n.[\alpha](\lambda x.n \times x))$$

(we allow ourselves to use integers and multiplication). It does not suffer from this problem, and corresponds to a function which, when applied to an argument n, turns into the function which multiplies by n. When we apply it to 3, it thus turns into the function which multiplies its argument (which is 3) by 3 and the result will actually be 9 as expected:

$$\begin{aligned} f\,3 &\longrightarrow \mu\beta.[\beta](\lambda n.[\beta](\lambda x.n \times x)3)3 \\ &\longrightarrow \mu\beta.[\beta][\beta](\lambda x.3 \times x)3 \\ &\longrightarrow \mu\beta.[\beta][\beta](3 \times 3) \end{aligned}$$

which is η-equivalent to 3×3 using the two η-conversion rules.

Another possible interpretation is that $\mu\alpha.t$ stores the current evaluation context and $[\alpha]u$ restores the evaluation context of the corresponding $\mu\alpha$ before executing u: it is as if the term t had never been executed. In the above example, it is as if f had directly been defined as $\lambda x.n \times x$.

4.6.3 Classical logic as a typing system. In order to type the $\lambda\mu$-calculus, we consider types of the form

$$A, B ::= X \mid A \to B \mid \bot$$

We also consider a Church variant of the calculus, where λ- and μ-abstracted variables are decorated by their types. The sequents are of the form

$$\Gamma \vdash t : A \mid \Delta$$

with t a term, A a type, and Γ and Δ contexts of the form

$$\Gamma = x_1 : B_1, \ldots, x_m : B_m \qquad \Delta = \alpha_1 : A_1, \ldots, \alpha_n : A_n$$

where the variables of Γ are regular ones, whereas those of Δ are control variables. Namely, Γ provides the type of the free variables of t as usual, whereas Δ gives the type of exceptions that might be raised. Finally, A is the type of the result of t, which might never be actually given if some exception is raised. In particular, a term of type $\bot$ is called a *command*: we know that it will never return a value, and thus necessarily raises some exception.

The typing rules for $\lambda\mu$-calculus are

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A \mid \Delta}(\text{ax})$$

$$\frac{\Gamma \vdash t : A \to B \mid \Delta \qquad \Gamma \vdash u : A \mid \Delta}{\Gamma \vdash t\,u : B \mid \Delta}(\to_{\text{E}}) \qquad \frac{\Gamma, x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x^A.t : A \to B \mid \Delta}(\to_{\text{I}})$$

$$\frac{\Gamma \vdash t : \bot \mid \Delta, \alpha : A, \Delta'}{\Gamma \vdash \mu\alpha^A.t : A \mid \Delta, \Delta'}(\bot_{\text{E}}) \qquad \frac{\Gamma \vdash t : A \mid \Delta, \alpha : A, \Delta'}{\Gamma \vdash [\alpha]t : \bot \mid \Delta, \alpha : A, \Delta'}(\bot_{\text{I}})$$

The rule $(\bot_{\text{E}})$ says that a term $\mu\alpha^A.t$ of type A is a command which raises some value of type A on α and the rule $(\bot_{\text{I}})$ says that a term $[\alpha]t$ is a command (of type $\bot$, not returning anything) and that the type A of the raised term t has to match the one expected for α.

Exercise 4.6.3.1. Show that the Pierce's law

$$((A \to B) \to A) \to A$$

is the type of the following term:

$$\lambda x^{(A \to B) \to A}.\mu\alpha^A.[\alpha](x\,(\lambda y^A.\mu\beta^B.[\alpha]y))$$

It can be shown [Par92, Par97] that this system has the expected properties which were detailed above for the case of simply-typed λ-calculus:

Theorem 4.6.3.2 (Subject reduction). If $\Gamma \vdash t : A \mid \Delta$ is derivable and $t \longrightarrow_\beta t'$ then $\Gamma \vdash t' : A \mid \Delta$ is also derivable.

Theorem 4.6.3.3 (Strong normalization). Typed $\lambda\mu$-terms are strongly normalizing.

If we erase the terms from the rules, we obtain the following presentation of classical logic:

$$\frac{}{\Gamma, A, \Gamma' \vdash A, \Delta}(\text{ax})$$

$$\frac{\Gamma \vdash A \Rightarrow B, \Delta \quad \Gamma \vdash A, \Delta}{\Gamma \vdash B, \Delta}(\Rightarrow_{\text{E}}) \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}(\Rightarrow_{\text{I}})$$

$$\frac{\Gamma \vdash \bot, \Delta, A, \Delta'}{\Gamma \vdash A, \Delta, \Delta'}(\bot_{\text{E}}) \qquad \frac{\Gamma \vdash A, \Delta, A, \Delta'}{\Gamma \vdash \bot, \Delta, A, \Delta'}(\bot_{\text{I}})$$

All the rules are the usual ones except for the rule ($\bot_{\text{I}}$) which combines weakening, contraction and exchange:

$$\frac{\dfrac{\dfrac{\Gamma \vdash A, \Delta, A, \Delta'}{\Gamma \vdash \Delta, A, A, \Delta'}(\text{xch})}{\Gamma \vdash \Delta, A, \Delta'}(\text{contr})}{\Gamma \vdash \bot, \Delta, A, \Delta'}(\text{wk})$$

The list of formulas in Δ (nor Γ) is not supposed to be commutative, and introduction and elimination rules always operate on the leftmost formula. During proof search we can however put another formula of Δ on the left using elimination and introduction rules for $\bot$, as shown on the left (and the corresponding typing derivation is figured on the right):

$$\frac{\dfrac{\Gamma \vdash B, A, \Delta, B, \Delta'}{\Gamma \vdash \bot, A, \Delta, B, \Delta}(\bot_{\text{I}})}{\Gamma \vdash A, \Delta, B, \Delta'}(\bot_{\text{E}}) \qquad \frac{\dfrac{\Gamma \vdash t : B \mid \alpha : A, \Delta, \beta : B, \Delta'}{\Gamma \vdash [\beta]t : \bot \mid A, \Delta, B, \Delta}(\bot_{\text{I}})}{\Gamma \vdash \mu\alpha^A.[\beta]t : A \mid \Delta, \beta : B, \Delta'}(\bot_{\text{E}})$$

Adding the usual rules for coproducts, we can show the excluded middle as follows in this settings

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A \mid \alpha : \neg A \vee A}(\text{ax})}{x : A \vdash \iota_{\text{r}}^{\neg A}(x) : \neg A \vee A \mid \alpha : \neg A \vee A}(\vee_{\text{I}}^{\text{r}})}{x : A \vdash [\alpha]\iota_{\text{r}}^{\neg A}(x) : \bot \mid \alpha : \neg A \vee A}(\bot_{\text{I}})}{\vdash \lambda x^A.[\alpha]\iota_{\text{r}}^{\neg A}(x) : \neg A \mid \alpha : \neg A \vee A}(\neg_{\text{I}})}{\vdash \iota_{\text{l}}^A(\lambda x^A.[\alpha]\iota_{\text{r}}^{\neg A}(x)) : A \vee B \mid \alpha : \neg A \vee A}(\vee_{\text{I}}^{\text{l}})}{\vdash [\alpha]\iota_{\text{l}}^A(\lambda x^A.[\alpha]\iota_{\text{r}}^{\neg A}(x)) : \bot \mid \alpha : \neg A \vee A}(\bot_{\text{I}})}{\vdash \mu\alpha^{\neg A \vee A}.[\alpha]\iota_{\text{l}}^A(\lambda x^A.[\alpha]\iota_{\text{r}}^{\neg A}(x)) : \neg A \vee A \mid}(\bot_{\text{E}})$$

In order to give a more concrete idea of this program, let us try to implement it in OCaml. Remember from section 1.5 that the empty type $\bot$ can be implemented as

```
type bot
```

and negation as

```
type 'a neg = 'a -> bot
```

From those, the above term proving excluded middle can roughly be translated as

```
let em () : (a neg, a) sum =
  let exception Alpha of (a neg, a) sum in
  try  Left (fun x -> raise (Alpha (Right x)))
  with Alpha x -> x
```

As explained above, this does not behave exactly as it should in OCaml, because exceptions are not properly scoped there...

4.6.4 A more symmetric calculus. The reduction rule for $(\mu\alpha.t)\,u$ in the $\lambda\mu$-calculus involves a slightly awkward substitution. In order to overcome this defect and reveal the symmetry of terms and environments, Curien and Herbelin have introduced a variant of the $\lambda\mu$-calculus called the $\bar{\lambda}\mu\tilde{\mu}$-*calculus* [CH00]. In this calculus there are three kinds of "terms":

$$\text{terms:} \qquad\qquad t ::= x \mid \lambda x.t \mid \mu\alpha.c$$
$$\text{environments:} \qquad\qquad e ::= \alpha \mid t \cdot e \mid \tilde{\mu}x.c$$
$$\text{commands:} \qquad\qquad c ::= \langle t \mid e \rangle$$

with reduction rules

$$\langle \lambda x.t \mid u \cdot e \rangle \longrightarrow \langle u \mid \tilde{\mu}x.\langle t \mid e \rangle \rangle$$
$$\langle \mu\alpha.c \mid e \rangle \longrightarrow c[e/\alpha]$$
$$\langle t \mid \tilde{\mu}x.c \rangle \longrightarrow c[t/x]$$

The typing judgments are of the three possible forms

$$\Gamma \vdash t : A \mid \Delta \qquad\qquad \Gamma \mid e : A \vdash \Delta \qquad\qquad c : (\Gamma \vdash \Delta)$$

and the rules are

$$\frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta}(\text{ax}_\text{L}) \qquad\qquad \frac{}{\Gamma, x : A \vdash x : A \mid \Delta}(\text{ax}_\text{R})$$

$$\frac{\Gamma \vdash t : A \mid \Delta \qquad \Gamma \mid e : B \vdash \Delta}{\Gamma, t \cdot e : A \to B \vdash \Delta}(\to_\text{L}) \qquad\qquad \frac{\Gamma, x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x.t, A \to B \mid \Delta}(\to_\text{R})$$

$$\frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}(\bot_\text{L}) \qquad\qquad \frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}(\bot_\text{R})$$

$$\frac{\Gamma \vdash t : A \mid \Delta \qquad \Gamma \mid e : A \vdash \Delta}{\langle t \mid e \rangle : (\Gamma \vdash \Delta)}$$

You are strongly encouraged to observe their beautiful symmetry and find out their meaning by yourself. In particular, Lafont's critical pair presented in section 2.5.4 corresponds to the fact that the following term can reduce in two different ways, showing that the calculus is not confluent (for good reasons!):

$$c[\tilde{\mu}x.d/\alpha] \longleftarrow \langle \mu\alpha.c \mid \tilde{\mu}x.d \rangle \longrightarrow d[\mu\alpha.c/x]$$

In particular, if α is not free in c and x is not free in d, c and d are convertible...

First-order logic

First-order logic is an extension of propositional logic where propositions are allowed to depend on terms over some fixed signature, and are then called *predicates*. For instance, *equality* can be encoded as a predicate $t = u$ which depends on two terms t and u. There are thus two worlds in play: the world of logic, where formulas live, and the world of data, where terms live. This logic is the one traditionally considered in mathematics (in particular, we will see that it can be used to formally state the axioms of set theory). Good introductions on the subject include [CK90, CL93].

We define first-order logic in section 5.1, present some well-known first-order theories in section 5.2, and detail the particular case of set theory in section 5.3 (including in the intuitionistic setting). Finally, the first-order unification algorithm is presented in section 5.4.

5.1 Definition

5.1.1 Signature. A *signature* Σ is a set of *function symbols* together with a function $a : \Sigma \to \mathbb{N}$ associating an *arity* to each symbol: f can be thought of as a formal operation with $a(f)$ inputs. In particular, symbols of arity 0 are called *constants*.

5.1.2 Terms. We suppose fixed an infinite countable set $\mathcal{X}$ of variables. Given a signature Σ, the set $\mathcal{T}_\Sigma$ of *terms* is the smallest set such that

– every variable is a term: $\mathcal{X} \subseteq \mathcal{T}_\Sigma$,

– terms are closed under operations: given $f \in \Sigma$ with $a(f) = n$ and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$, we have $f(t_1, \ldots, t_n) \in \mathcal{T}_\Sigma$.

This can also be stated as the fact that terms are generated by the grammar

$$t ::= x \mid f(t_1, \ldots, t_n)$$

where x is a variable, f is a term of arity n and the t_i are terms. We often implicitly suppose fixed a signature and simply write $\mathcal{T}$ instead of $\mathcal{T}_\Sigma$.

Example 5.1.2.1. Consider the signature $\Sigma = \{+ : 2, 0 : 0\}$. This notation means that it contains two functions symbols $+$ and 0, whose arities are respectively $a(+) = 2$ and $a(0) = 0$. Examples of terms over this signature are

$$+(x, 0()) \qquad +(+(x, x), +(y, 0())) \qquad +(0(), 0())$$

In the following, we generally omit parenthesis for constants, e.g. write 0 instead of $0()$.

Given a term t, its set of *subterms* $\mathrm{ST}(t)$ is defined by induction on t by

$$\mathrm{ST}(x) = \{x\}$$

$$\mathrm{ST}(f(t_1, \ldots, t_n)) = \{f(t_1, \ldots, t_n)\} \cup \bigcup_{1 \leqslant i \leqslant n} \mathrm{ST}(t_i)$$

We say that u is a *subterm* of t when $u \in \mathrm{ST}(t)$, it is a *strict subterm* when it is distinct from t.

5.1.3 Substitutions. A *substitution* is a function $\sigma : \mathcal{X} \to \mathcal{T}$ such that the set $\{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite. Given a term t, we write $t[\sigma]$ for the term obtained from t by replacing every variable x by $\sigma(x)$:

$$x[\sigma] = \sigma(x) \qquad (f(t_1, \ldots, t_n))[\sigma] = f(t_1[\sigma], \ldots, t_n[\sigma])$$

We sometimes write $\sigma = [t_1/x_1, \ldots, t_n/x_n]$ for the substitution such that $\sigma(x_i) = t_i$ and $\sigma(x) = x$ for $x \neq x_i$ for every $1 \leqslant i \leqslant n$. A *renaming* is a substitution such that the term $\sigma(x)$ is a variable, for every variable x.

5.1.4 Formulas. We suppose fixed a set $\mathcal{P}$ of *predicates* (also sometimes called *relation symbols*) together with a function $a : \mathcal{P} \to \mathbb{N}$ associating an *arity* to each predicate. A *formula A* is an expression generated by the grammar

$$A, B ::= P(t_1, \ldots, t_n) \mid A \Rightarrow B \mid A \wedge B \mid \top \mid A \vee B \mid \bot \mid \neg A \mid \forall x.A \mid \exists x.A$$

where P is a predicate of arity n, the t_i are terms, $x \in \mathcal{X}$ is a term variable and A and B are formulas. The quantifications bind the less tightly, e.g. $\forall x.A \wedge B$ is implicitly bracketed as $\forall x.(A \wedge B)$ and not $(\forall x.A) \wedge B$. Note that the definition of formulas depends both on the considered signature Σ and the considered set $\mathcal{P}$ of predicates: we sometimes say a formula *on $(\Sigma, \mathcal{P})$* to make this precise, although we generally leave it implicit.

Example 5.1.4.1. Consider the signature $\Sigma = \{\times : 2, 1 : 0\}$, which means that we have two function symbols "$\times$" and "1", with $\times$ of arity 2 and 1 of arity 0. We also suppose that $\mathcal{P}$ contains a predicate $=$ of arity 2. We have the formula

$$\forall x.\exists y.(x \times y = 1 \wedge y \times x = 1)$$

which expresses that every element admits an inverse.

Example 5.1.4.2. With a predicate D of arity one, the *drinker formula* is

$$\exists x.(D(x) \Rightarrow (\forall y.D(y)))$$

The name of this formula comes from the following interpretation. If we see terms as people in a pub and consider that $D(t)$ holds when t drinks, it can be read as:

<div style="text-align:center">

There is someone in the pub such that,
if he is drinking, then everyone in the pub is drinking.

</div>

We will see in example 5.1.7.1 that this formula is classically true, but that it is not intuitionistically so.

First order logic is an extension of propositional logic in the following sense. Consider the empty signature $\Sigma = \emptyset$ and the set $\mathcal{P} = \mathcal{X}$ consisting of all propositional variables, seen as predicates of arity 0. Then a propositional formula, such as $X \vee \neg Y$, corresponds precisely to a first order formula, such as $X() \vee \neg Y()$.

5.1.5 Bound and free variables. In a formula of the form $\forall x.A$ or $\exists x.A$, the variable x is said to be *bound* in A. This means that the name of the variable x does not really matter and we could have renamed it to some other variable name, without changing the formula. We thus implicitly consider formulas up to proper (or *capture avoiding*) renaming of variables (by "proper", we mean here that we should take care of not renaming a variable to some already bound variable name). For instance, we consider that the two formulas

$$\forall x.\exists y.x + y = x \qquad \text{and} \qquad \forall z.\exists y.z + y = z$$

are the same (the second is obtained from the first by renaming x to z), but they are different from the formula

$$\forall x.\exists x.x + x = x$$

obtained by an "improper" renaming of y into x which was already bound. Such a mechanism for renaming bound variables is detailed in section 3.1, for the λ-calculus.

A variable which is not bound is said to be *free* and we write $\mathrm{FV}(A)$ for the set of free variables of a formula A. This is formally defined by

$$\mathrm{FV}(P(t_1,\ldots,t_n)) = \mathrm{FV}(t_1) \cup \ldots \cup \mathrm{FV}(t_n)$$
$$\mathrm{FV}(A \Rightarrow B) = \mathrm{FV}(A \times B) = \mathrm{FV}(A + B) = \mathrm{FV}(A) \cup \mathrm{FV}(B)$$
$$\mathrm{FV}(\top) = \mathrm{FV}(\bot) = \emptyset$$
$$\mathrm{FV}(\neg A) = \mathrm{FV}(A)$$
$$\mathrm{FV}(\forall x.A) = \mathrm{FV}(\exists x.A) = \mathrm{FV}(A) \setminus \{x\}$$

where, given a term t, we write $FV(t)$ for the set of all the variables occurring in t. A formula A is *closed* when it has no free variables, i.e. $\mathrm{FV}(A) = \emptyset$. We sometimes write

$$A(x_1,\ldots,x_n)$$

for a formula A whose free variables are among $x_1,\ldots,x_n$. In this case, we write $A(t_1,\ldots,t_n)$ instead of $A[t_1/x_1,\ldots,t_n/x_n]$.

Given a formula A and a term t, we write $A[t/x]$ for the formula A where all the free occurrences of x have been *substituted* by t avoiding captures, i.e. we suppose that all bound variables are different from the variables of t. For instance, with A being
$$(\exists y.x + x = y) \vee (\exists x.x = y)$$
we have that $A[z + z/x]$ is

$$(\exists y.(z + z) + (z + z) = y) \vee (\exists x.x = y)$$

but in order to compute $A[y + y/x]$, we have to rename the bound variable y (say, to z) and the result will be

$$(\exists z.(y + y) + (y + y) = z) \vee (\exists x.x = y)$$

and not

$$(\exists y.(y + y) + (y + y) = y) \vee (\exists x.x = y)$$

5.1.6 Natural deduction rules. The rules for first order logic in intuitionistic natural deduction are the usual ones (see figure 2.1) together with the following introduction and elimination rules for universal and existential quantification:

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[t/x]} \; (\forall_{\mathrm{E}}) \qquad\qquad\qquad \frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A} \; (\forall_{\mathrm{I}})$$

$$\frac{\Gamma \vdash \exists x.A \qquad \Gamma, A \vdash B}{\Gamma \vdash B} \; (\exists_{\mathrm{E}}) \qquad\qquad \frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x.A} \; (\exists_{\mathrm{I}})$$

These rules are subject to the following (important) side conditions:

– in $(\forall_{\mathrm{I}})$, we suppose $x \notin \mathrm{FV}(\Gamma)$,

– in $(\exists_{\mathrm{E}})$, we suppose $x \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(B)$.

where, given a context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, we have

$$\mathrm{FV}(\Gamma) = \bigcup_{i=1}^{n} \mathrm{FV}(A_i)$$

Example 5.1.6.1. We have

$$\frac{\displaystyle \frac{}{\forall x.\neg A, \exists x.A \vdash \exists x.A} \; (\mathrm{ax}) \qquad \frac{\displaystyle \frac{\displaystyle \frac{}{\forall x.\neg A, \exists x.A, A \vdash \forall x.\neg A} \; (\mathrm{ax})}{\forall x.\neg A, \exists x.A, A \vdash \neg A} \; (\forall_{\mathrm{E}}) \qquad \frac{}{\forall x.\neg A, \exists x.A, A \vdash A} \; (\mathrm{ax})}{\displaystyle \frac{\forall x.\neg A, \exists x.A, A \vdash \bot}{\forall x.\neg A, \exists x.A \vdash \bot}} \; (\neg_{\mathrm{E}}) }{\displaystyle \frac{\forall x.\neg A \vdash \neg(\exists x.A)}{\vdash (\forall x.\neg A) \Rightarrow \neg(\exists x.A)}}$$

Remark 5.1.6.2. The side conditions avoid clearly problematic proofs such as

$$\frac{\displaystyle \frac{\displaystyle \frac{\displaystyle \frac{}{A(x) \vdash A(x)} \; (\mathrm{ax})}{A(x) \vdash \forall x.A(x)} \; (\forall_{\mathrm{I}})}{\displaystyle \frac{\vdash A(x) \Rightarrow \forall x.A(x)}{\vdash \forall x.(A(x) \Rightarrow \forall x.A(x))} \; (\forall_{\mathrm{I}})} \; (\Rightarrow_{\mathrm{I}})}{\vdash A(t) \Rightarrow \forall x.A(x)} \; (\forall_{\mathrm{E}})$$

which can be read as: if the formula A holds for some term t then it holds for any term. The problematic rule is the $(\forall_{\mathrm{I}})$ just after the (ax) rule: x is not fresh.

Properties of the calculus. We do not detail this here, but the usual properties of natural deduction generalize to first order logic. In particular, the structural rules (contraction, exchange, weakening) are admissible, see section 2.2.7. We will also see in section 5.1.9 that cuts can be eliminated.

5.1.7 Classical first order logic. Following section 2.5, *classical* first order logic, is the system obtained from the above one by adding one of the following rules

$$\frac{}{\Gamma \vdash \neg A \vee A} \ (\text{lem}) \qquad \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A} \ (\neg\neg_\text{E}) \qquad \frac{\Gamma, \neg A \vdash A}{\Gamma \vdash A} \ (\text{raa})$$

implementing the excluded middle, the elimination of double negation or Clavius' law (we could also have added any of the axioms of theorem 2.5.1.1).

Example 5.1.7.1. A typical formula which is provable in classical logic (and not in intuitionistic logic) is

$$A = \exists x.(D(x) \Rightarrow (\forall y.D(y)))$$

already presented in example 5.1.4.2. A proof is the following:

$$\frac{\frac{\dfrac{}{\ldots, \neg D(y) \vdash \neg D(y)} \ (\text{ax}) \quad \dfrac{}{\ldots, D(y) \vdash D(y)} \ (\text{ax})}{\dfrac{\neg A, D(x), \neg D(y), D(y) \vdash \bot}{\dfrac{\neg A, D(x), \neg D(y), D(y) \vdash \forall y.D(y)}{\dfrac{\neg A, D(x), \neg D(y) \vdash D(y) \Rightarrow (\forall y.D(y))}{\dfrac{\neg A, D(x), \neg D(y) \vdash \exists x.(D(x) \Rightarrow (\forall y.D(y)))}{\dfrac{\neg A, D(x), \neg D(y) \vdash \bot}{\dfrac{\neg A, D(x) \vdash \neg\neg D(y)}{\dfrac{\neg A, D(x) \vdash D(y)}{\dfrac{\neg A, D(x) \vdash \forall y.D(y)}{\dfrac{\neg A \vdash D(x) \Rightarrow (\forall y.D(y))}{\dfrac{\neg A \vdash \exists x.(D(x) \Rightarrow (\forall y.D(y)))}{\vdash \exists x.(D(x) \Rightarrow (\forall y.D(y)))} \ (\text{raa})} \ (\exists_\text{I})} \ (\Rightarrow_\text{I})} \ (\forall_\text{I})} \ (\neg\neg_\text{E})} \ (\neg_\text{I})} \ (\neg_\text{E})} \ (\exists_\text{I})} \ (\Rightarrow_\text{I})} \ (\bot_\text{E})} \ (\neg_\text{E})}$$

If we interpret x as ranging over the people present in a pub, and the predicate $D(x)$ as "x drinks" this formula states that there is a "universal drinker", i.e. somebody such that if he drinks then everybody drinks. We can imagine why this formula cannot be proved intuitionistically: if it was so, we should be able to come up with an explicit name for this guy, see theorem 5.1.9.3, which seems impossible in absence of further information on the pub. We do not actually prove that there exists x such that $D(x)$, which would require us to come up with an explicit witness for x, but only show that it cannot be the case that there is no x satisfying D, which is enough to conclude by double negation elimination.

Example 5.1.7.2. Another formula provable in classical logic is the formula

$$\neg(\forall x.\neg A(x)) \Rightarrow \exists x.A(x)$$

which states that if it is not the case that every element x does not satisfy $A(x)$, then we can actually produce an element which satisfies $A(x)$. It can be proved

as follows:

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{
        \cfrac{
          \cfrac{
            \cfrac{\ \ }{\dots \vdash \neg\exists x.A(x)}\ (\text{ax})
            \qquad
            \cfrac{\cfrac{\ \ }{\dots \vdash A(x)}\ (\text{ax})}{\dots \vdash \exists x.A(x)}\ (\exists_{\mathrm{I}})
          }{\neg\forall x.\neg A(x), \neg\exists x.A(x), A(x) \vdash \bot}\ (\neg_{\mathrm{E}})
        }{\neg\forall x.\neg A(x), \neg\exists x.A(x) \vdash \neg A(x)}\ (\neg_{\mathrm{I}})
      }{\neg\forall x.\neg A(x), \neg\exists x.A(x) \vdash \forall x.\neg A(x)}\ (\forall_{\mathrm{I}})
    }{\cfrac{\ \ }{\dots \vdash \neg\forall x.\neg A(x)}\ (\text{ax}) \qquad \neg\forall x.\neg A(x), \neg\exists x.A(x) \vdash \bot}\ (\neg_{\mathrm{E}})
  }{\neg\forall x.\neg A(x) \vdash \neg\neg\exists x.A(x)}\ (\neg_{\mathrm{I}})
  }{\neg\forall x.\neg A(x) \vdash \exists x.A(x)}\ (\neg\neg_{\mathrm{E}})
}{\vdash \neg(\forall x.\neg A(x)) \Rightarrow \exists x.A(x)}\ (\Rightarrow_{\mathrm{I}})
$$

As in example 5.1.7.1, it is enough to show that it is not the case that there is no x satisfying A.

Exercise 5.1.7.3. Another proof for the drinker formula of example 5.1.7.1 is the following. We have two possibilities for the pub:

- either everybody drinks: in this case, we can take anybody as the universal drinker,

- otherwise, there is someone who does not drink: we can take him as universal drinker.

Formalize this reasoning in natural deduction.

De Morgan laws. In addition to the equivalences already shown in section 2.5.5, the following *de Morgan laws* hold in classical first-order logic:

$$(\forall x.A) \wedge B \Leftrightarrow \forall x.(A \wedge B) \qquad\qquad B \wedge (\forall x.A) \Leftrightarrow \forall x.(B \wedge A)$$
$$(\forall x.A) \vee B \Leftrightarrow \forall x.(A \vee B) \qquad\qquad B \vee (\forall x.A) \Leftrightarrow \forall x.(B \vee A)$$
$$(\forall x.A) \Rightarrow B \Leftrightarrow \exists x.(A \Rightarrow B) \qquad\qquad B \Rightarrow (\forall x.A) \Leftrightarrow \forall x.(B \Rightarrow A)$$
$$(\exists x.A) \wedge B \Leftrightarrow \exists x.(A \wedge B) \qquad\qquad B \wedge (\exists x.A) \Leftrightarrow \exists x.(B \wedge A)$$
$$(\exists x.A) \vee B \Leftrightarrow \exists x.(A \vee B) \qquad\qquad B \vee (\exists x.A) \Leftrightarrow \exists x.(B \vee A)$$
$$(\exists x.A) \Rightarrow B \Leftrightarrow \forall x.(A \Rightarrow B) \qquad\qquad B \Rightarrow (\exists x.A) \Leftrightarrow \exists x.(B \Rightarrow A)$$

whenever $x \notin \mathrm{FV}(B)$, as well as

$$\neg(\forall x.A) \Leftrightarrow \exists x.\neg A \qquad\qquad \neg(\exists x.A) \Leftrightarrow \forall x.\neg A$$

Prenex form. A formula P is in *prenex form* when it is of the form

$$P ::= \forall x.P \mid \exists x.P \mid A$$

where A is a formula which does not contain any first-order quantification: a formula in prenex form thus consists in a bunch of universal and existential quantifications over a formula without quantifications. By using the above de Morgan laws from left to right, one can show that

Lemma 5.1.7.4. Every formula is equivalent to a formula in prenex form.

Example 5.1.7.5. The formula of example 5.1.7.2 can be put into prenex form as follows:

$$\neg(\forall x.\neg A(x)) \Rightarrow \exists x.A(x) \quad \rightsquigarrow \quad (\exists x.\neg\neg A(x)) \Rightarrow \exists x.A(x)$$
$$\rightsquigarrow \quad \forall x.\neg\neg A(x) \Rightarrow \exists x.A(x)$$
$$= \quad \forall x.\neg\neg A(x) \Rightarrow \exists y.A(y)$$
$$\rightsquigarrow \quad \forall x.\exists y.\neg\neg A(x) \Rightarrow A(y)$$

More de Morgan laws. In addition to the above equivalences, we also have

$$\forall x.(A \wedge B) \Leftrightarrow (\forall x.A) \wedge (\forall x.B) \qquad \exists x.(A \vee B) \Leftrightarrow (\exists x.A) \vee (\exists x.B)$$
$$\forall x.\top \Leftrightarrow \top \qquad\qquad\qquad \exists x.\bot \Leftrightarrow \bot$$

5.1.8 Sequent calculus rules. The rules for first-order quantifiers in classical sequent calculus are

$$\frac{\Gamma, \forall x.A, A[t/x] \vdash \Delta}{\Gamma, \forall x.A \vdash \Delta}(\forall_{\mathrm{L}}) \qquad\qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash \forall x.A, \Delta}(\forall_{\mathrm{R}})$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, \exists x.A \vdash \Delta}(\exists_{\mathrm{L}}) \qquad\qquad \frac{\Gamma \vdash A[t/x], \exists x.A, \Delta}{\Gamma \vdash \exists x.A, \Delta}(\exists_{\mathrm{R}})$$

with the side condition for $(\forall_{\mathrm{R}})$ and $(\exists_{\mathrm{L}})$ that $x \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(\Delta)$. Intuitionistic rules are obtained, as usual, by restricting to sequents with one formula on the right:

$$\frac{\Gamma, \forall x.A, A[t/x] \vdash B}{\Gamma, \forall x.A \vdash B}(\forall_{\mathrm{L}}) \qquad\qquad \frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A}(\forall_{\mathrm{R}})$$

$$\frac{\Gamma, A \vdash B}{\Gamma, \exists x.A \vdash B}(\exists_{\mathrm{L}}) \qquad\qquad \frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x.A}(\exists_{\mathrm{R}})$$

with the expected side conditions for $(\forall_{\mathrm{R}})$ a,d $(\exists_{\mathrm{L}})$.

Remark 5.1.8.1. In the rules $(\forall_{\mathrm{L}})$ and $(\exists_{\mathrm{R}})$, we have been careful to keep a copy of the hypothesis: with this formulation, contraction is admissible.

Example 5.1.8.2. The drinker formula from example 5.1.4.2 can be proved classically by

$$\frac{\dfrac{\overline{D(x), D(y) \vdash D(y), \forall y.D(y), \exists x.(D(x) \Rightarrow (\forall y.D(y)))}^{(\mathrm{ax})}}{\dfrac{D(x) \vdash D(y), D(y) \Rightarrow (\forall y.D(y)), \exists x.(D(x) \Rightarrow (\forall y.D(y)))}{\dfrac{D(x) \vdash D(y), \exists x.(D(x) \Rightarrow (\forall y.D(y)))}{\dfrac{D(x) \vdash \forall y.D(y), \exists x.(D(x) \Rightarrow (\forall y.D(y)))}{\dfrac{\vdash D(x) \Rightarrow (\forall y.D(y)), \exists x.(D(x) \Rightarrow (\forall y.D(y)))}{\vdash \exists x.(D(x) \Rightarrow (\forall y.D(y)))}(\exists_{\mathrm{R}})}(\Rightarrow_{\mathrm{R}})}(\forall_{\mathrm{R}})}(\exists_{\mathrm{R}})}(\Rightarrow_{\mathrm{R}})}$$

As noted in the previous remark, we need to use the proved formula twice, and it is thus crucial that we keep a copy of it in the rule $(\exists_{\mathrm{R}})$ at the bottom.

5.1.9 Cut elimination. The properties and proof techniques developed in section 2.3 extend to first order natural deduction, allowing to prove that it has the cut elimination property:

Theorem 5.1.9.1. A sequent $\Gamma \vdash A$ admits a proof if and only if it admits a cut free proof.

In the cut elimination procedure, there are two new cases, which can be handled as follows:

$$
\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash A(x)}}{\cfrac{\Gamma \vdash \forall x.A(x)}{\Gamma \vdash A(t)}} \ (\forall_{\mathrm{I}})}{\Gamma \vdash A(t)} \ (\forall_{\mathrm{E}}) \quad \rightsquigarrow \quad \cfrac{\pi[t/x]}{\Gamma \vdash A(t)}
$$

$$
\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash A(t)}}{\Gamma \vdash \exists x.A(x)} \ (\exists_{\mathrm{I}}) \qquad \cfrac{\pi'}{\Gamma, A(x) \vdash B}}{\Gamma \vdash B} \ (\exists_{\mathrm{E}}) \quad \rightsquigarrow \quad \cfrac{\pi'[t/x][\pi/A]}{\Gamma \vdash B}
$$

Above, $\pi[t/x]$ stands for the proof π where all the free occurrences of the variable x have been replaced by the term t (details left to the reader). As in the case of propositional logic, it can be shown that a proof of a formula in an empty context necessarily ends with an introduction rule (proposition 2.3.3.2) and thus deduce (as in theorem 2.3.4.2):

Theorem 5.1.9.2 (Consistency). First order (intuitionistic or classical) natural deduction is consistent: there is no proof of $\vdash \bot$.

Another important consequence is that that the logic has the *existence property*: if we can prove that there exists a term satisfying some property, then we can actually construct such a term:

Theorem 5.1.9.3 (Existence property). A formula of the form $\exists x.A$ is provable in intuitionistic first order natural deduction if and only if there exists a term t such that $A[t/x]$ is provable.

Proof. For the left-to-right implication, if we have a proof of $\exists x.A$ then, by theorem 5.1.9.1, we have a cut-free one which, by proposition 2.3.3.2, ends with an introduction rule, i.e. is of the form

$$
\cfrac{\cfrac{\pi}{\vdash A[t/x]}}{\vdash \exists x.A} \ (\exists_{\mathrm{I}})
$$

We therefore have a proof π of $A[t/x]$ for some term t. The right-to-left implication is given by an application of the rule $(\exists_{\mathrm{I}})$. $\qquad \square$

In contrast, we do not expect this property to hold in classical logic. For instance, consider the drinker formula of example 5.1.7.2. We can feel that the proof we have given is not constructive: there is no way of determining who is the drinker in general (i.e. without performing a reasoning specific to the bar in which we currently are).

5.1.10 Eigenvariables. The logic as we have presented it (which is the way it is traditionally presented) suffers from a defect: in a given sequent, we do not know the first order variables which are used. This is the subtle cause of some surprising proofs. For instance, we can prove that there always exists a term, whereas we would expect that the empty set is a perfectly reasonable way of interpreting logic in the case where the signature is empty for instance:

$$\dfrac{\dfrac{}{\vdash \top} \ (\top_{\mathrm{I}})}{\vdash \exists x.\top} \ (\exists_{\mathrm{I}})$$

Note that in the premise of the $(\exists_{\mathrm{I}})$, we use the fact that $\top = \top[x/x]$, i.e. we use x as witness for the existence. A variation on the previous example is the following proof, which expresses the fact that if a property A is satisfied for every term x, then we can exhibit a term satisfying A. Again, we would have expected that this is not true if there is no element in the model, and moreover, this does not feel very constructive:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x.A \vdash \forall x.A} \ (\mathrm{ax})}{\forall x.A \vdash A} \ (\forall_{\mathrm{E}})}{\forall x.A \vdash \exists x.A} \ (\exists_{\mathrm{I}})}{\vdash (\forall x.A) \Rightarrow \exists x.A} \ (\Rightarrow_{\mathrm{I}})$$

Here also, in the premise of the $(\exists_{\mathrm{I}})$ rule, we use the fact that $A = A[x/x]$, i.e. we use x as witness for the existence. We will see in section 5.2.3 that this is the reason why models are usually supposed to be non-empty, while there is no good reason to exclude this particular case.

In order to fix that, we should keep track of the variables which are declared in the context, which are sometimes called *eigenvariables*. This can be done by adding a new context Ξ to our sequents, which is a list of first order variables which are declared. We thus consider sequents of the form

$$\Xi \mid \Gamma \vdash A$$

the vertical bar being there to mark the delimitation between the context of eigenvariables and the traditional context. The rules for logical connectives simply "propagate" the new context Ξ, e.g. the rules for conjunction become

$$\dfrac{\Xi \mid \Gamma \vdash A \wedge B}{\Xi \mid \Gamma \vdash A} \ (\wedge_{\mathrm{E}}^{\mathrm{l}}) \qquad \dfrac{\Xi \mid \Gamma \vdash A \wedge B}{\Xi \mid \Gamma \vdash B} \ (\wedge_{\mathrm{E}}^{\mathrm{r}}) \qquad \dfrac{\Xi \mid \Gamma \vdash A \quad \Xi \mid \Gamma \vdash B}{\Xi \mid \Gamma \vdash A \wedge B} \ (\wedge_{\mathrm{I}})$$

More interestingly, the rules for first order quantifiers become

$$\dfrac{\Xi \mid \Gamma \vdash \forall x.A}{\Xi \mid \Gamma \vdash A[t/x]} \ (\forall_{\mathrm{E}}) \qquad\qquad \dfrac{\Xi, x \mid \Gamma \vdash A}{\Xi \mid \Gamma \vdash \forall x.A} \ (\forall_{\mathrm{I}})$$

$$\dfrac{\Xi \mid \Gamma \vdash \exists x.A \quad \Xi, x \mid \Gamma, A \vdash B}{\Xi \mid \Gamma \vdash B} \ (\exists_{\mathrm{E}}) \qquad\qquad \dfrac{\Xi \mid \Gamma \vdash A[t/x]}{\Xi \mid \Gamma \vdash \exists x.A} \ (\exists_{\mathrm{I}})$$

where we suppose

– $x \notin \Xi$ in ($\forall_I$) and ($\exists_E$),

– $FV(t) \subseteq \Xi$ in ($\forall_E$) and ($\exists_I$).

Finally, the axiom rule and the truth introduction rule become

$$\frac{\Xi \mid \Gamma, A, \Gamma' \vdash}{\Xi \mid \Gamma, A, \Gamma' \vdash A} \text{ (ax)} \qquad\qquad \frac{\Xi \mid \Gamma \vdash}{\Xi \mid \Gamma \vdash \top} \text{ ($\top_I$)}$$

where $\Xi \mid \Gamma \vdash$ is a notation to mean that we suppose $FV(\Gamma) \subseteq \Xi$. Supposing this for these two rules (which are the only two without premise) is enough to ensure that whenever we prove a sequent $\Xi \mid \Gamma \vdash A$, we will always have $FV(\Gamma) \cup FV(A) \subseteq \Xi$ (it is easy to check that the inference rules preserve this invariant).

Example 5.1.10.1. We can still prove $(\forall x.A) \Rightarrow \forall x.A$ in this new system:

$$\frac{\dfrac{\dfrac{}{x \mid \forall x.A \vdash \forall x.A} \text{ (ax)}}{\dfrac{x \mid \forall x.A \vdash A[x/x]}{\dfrac{\mid \forall x.A \vdash \forall x.A}{\mid \vdash (\forall x.A) \Rightarrow \forall x.A} \text{ ($\Rightarrow_I$)}} \text{ ($\forall_I$)}}{} \text{ ($\forall_E$)}}$$

Example 5.1.10.2. We cannot prove $\exists x.\top$ in this system. In particular, the proof

$$\frac{\dfrac{}{\mid \vdash \top[x/x]} \text{ ($\top_I$)}}{\mid \vdash \exists x.\top} \text{ ($\exists_I$)}$$

is not valid because the side condition is not satisfied for the rule ($\exists_I$).

Exercise 5.1.10.3. Show that the formula $(\forall x.\bot) \Rightarrow \bot$ is provable with traditional rules, but not with the rules presented in this section.

5.1.11 Curry-Howard. The Curry-Howard correspondence can be extended to first-order logic, following the intuition that

– a proof of $\forall x.A$ should be a function which, when applied to a term t, returns a proof that A is valid for this term,

– a proof of $\exists x.A$ should be a pair consisting of a term t and a proof that A is valid for this term.

Expressions. We begin with the language for proofs introduced in chapter 4, the simply typed λ-calculus. In this section, we call its terms *expressions* in order not to confuse them with first order terms, and write e for an expression. The syntax for expressions is thus

$$e, e' ::= \lambda x^A.e \mid e\,e' \mid \ldots$$

In order to account for first order logic, we extend expressions with the following constructions:

$$e ::= \ldots \mid \overset{\forall}{\lambda} x.e \mid e\,t \mid \langle t, e \rangle \mid \text{unpair}(e, xy \mapsto e')$$

The newly added constructions are

- $\overset{\forall}{\lambda}x.e$: a function taking a term as argument x and returning an expression e,

- $e\,t$: the application of an expression (typically a function as above) to a term t,

- $\langle t, e \rangle$: a pair consisting of a term t and an expression e,

- $\mathrm{unpair}(e, xy \mapsto e')$: the extraction of the components x and y of a pair e for use in an expression e', which would be written in a syntax closer to OCaml

$$\texttt{let } \langle x, y \rangle \texttt{ = } e \texttt{ in } e'$$

We insist on the fact that there are two kinds of abstractions, respectively written λ and $\overset{\forall}{\lambda}$, and two kind of applications, which are distinct constructions. Similarly, for products, there are two kinds of pairings and of eliminators. Although they behave similarly, they are entirely distinct constructions. However, we will be able to unify those constructions when going to dependent types in chapter 8: there will be one kind of abstraction (resp. pairing) which covers both cases.

Typing rules. The associated typing rules are

$$\frac{\Gamma \vdash e : \forall x.A}{\Gamma \vdash e\,t : A[t/x]}(\forall_{\mathrm{E}}) \qquad\qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \overset{\forall}{\lambda}x.e : \forall x.A}(\forall_{\mathrm{I}})$$

$$\frac{\Gamma \vdash e : \exists x.A \qquad \Gamma, y : A \vdash e' : B}{\Gamma \vdash \mathrm{unpair}(e, xy \mapsto e') : B}(\exists_{\mathrm{E}}) \qquad \frac{\Gamma \vdash e : A[t/x]}{\Gamma \vdash \langle t, e \rangle : \exists x.A}(\exists_{\mathrm{I}})$$

and can be read as follows:

- $(\forall_{\mathrm{I}})$: a proof of $\forall x.A$ is a function which takes a term x as argument and returns a proof of A,

- $(\forall_{\mathrm{E}})$: using a proof of $\forall x.A$ consists in applying it to a term t,

- $(\exists_{\mathrm{I}})$: a proof of $\exists x.A(x)$ is a pair consisting of a term t and a proof that $A(t)$ is satisfied,

- $(\exists_{\mathrm{E}})$: we can use a proof of $\exists x.A$ by extracting its components.

Example 5.1.11.1. Consider again the derivation of example 5.1.6.1. It can be decorated with expressions as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{\overline{f : \forall x.\neg A, e : \exists x.A \vdash e : \exists x.A}\,(\mathrm{ax})}{} \quad \cfrac{\cfrac{\overline{f : \forall x.\neg A, e : \exists x.A, a : A \vdash f : \forall x.\neg A}\,(\mathrm{ax})}{f : \forall x.\neg A, e : \exists x.A, a : A \vdash f\,x : \neg A}(\forall_{\mathrm{E}}) \quad \overline{f : \forall x.\neg A, e : \exists x.A, a : A \vdash a : A}\,(\mathrm{ax})}{f : \forall x.\neg A, e : \exists x.A, a : A \vdash f\,x\,a : \bot}(\neg_{\mathrm{E}})}{f : \forall x.\neg A, e : \exists x.A \vdash \mathrm{unpair}(e, xa \mapsto f\,x\,a) : \bot}(\exists_{\mathrm{E}})}{f : \forall x.\neg A \vdash \lambda e^{\exists x.A}.\,\mathrm{unpair}(e, xa \mapsto f\,x\,a) : \neg(\exists x.A)}(\neg_{\mathrm{I}})}{\vdash \lambda f^{\forall x.\neg A}.\lambda e^{\exists x.A}.\,\mathrm{unpair}(e, xa \mapsto f\,x\,a) : (\forall x.\neg A) \Rightarrow \neg(\exists x.A)}(\Rightarrow_{\mathrm{I}})$$

The corresponding proof term is thus

$$\lambda f^{\forall x.\neg A}.\lambda e^{\exists x.A}.\,\mathrm{unpair}(e, xa \mapsto f\,x\,a)$$

This function takes two arguments:

– f of type $\forall x.A \to \bot$, and

– e of type $\exists x.A$

and produces a value of type $\bot$ by extracting from e a term x and a proof a of $A(t)$, and applying f to x and a.

Reduction. As usual, the β-reduction rules correspond to cut-elimination steps:

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash e : A}}{\Gamma \vdash \breve{\forall}x.e : \forall x.A}\,(\forall_{\mathrm{I}})}{\Gamma \vdash (\breve{\forall}x.e)\,t : A[t/x]}\,(\forall_{\mathrm{E}}) \qquad \rightsquigarrow \qquad \cfrac{\pi[t/x]}{\Gamma \vdash e[t/x] : A[t/x]}$$

i.e.

$$(\breve{\forall}x.e)\,t \quad \longrightarrow_\beta \quad e[t/x]$$

and

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash e : A[t/x]}}{\Gamma \vdash \langle t, e \rangle : \exists x.A}\,(\exists_{\mathrm{I}}) \qquad \cfrac{\pi'}{\Gamma, y : A \vdash e' : B}}{\Gamma \vdash \mathrm{unpair}(\langle t, e \rangle, xy \mapsto e') : B}\,(\exists_{\mathrm{E}}) \qquad \rightsquigarrow \qquad \cfrac{\pi'[t/x][\pi/A]}{\Gamma \vdash e'[t/x, e/y] : B}$$

i.e.

$$\mathrm{unpair}(\langle t, e \rangle, xy \mapsto e') \quad \longrightarrow_\beta \quad e'[t/x, e/y]$$

where $\pi[t/x]$ is the proof obtained from π by replacing all free occurrences of x by t (details left to the reader). Similarly, η-reduction rules correspond to eliminate dual of cuts:

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash e : \forall x.A}}{\Gamma \vdash e\,x : A[x/x]}\,(\forall_{\mathrm{E}})}{\Gamma \vdash \breve{\forall}x.e\,x : \forall x.A}\,(\forall_{\mathrm{I}}) \qquad \rightsquigarrow \qquad \cfrac{\pi}{\Gamma \vdash e : \forall x.A}$$

i.e.

$$\breve{\forall}x.e\,x \quad \longrightarrow_\eta \quad e$$

and

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash e : \exists x.A} \qquad \cfrac{}{\Gamma, y : A \vdash y : A}\,(\mathrm{ax})}{\Gamma \vdash \mathrm{unpair}(e, xy \mapsto y) : A}\,(\exists_{\mathrm{E}})}{\Gamma \vdash \langle x, \mathrm{unpair}(e, xy \mapsto y) \rangle : \exists x.A}\,(\exists_{\mathrm{I}}) \qquad \rightsquigarrow \qquad \cfrac{\pi}{\Gamma \vdash e : \exists x.A}$$

i.e.

$$\langle x, \mathrm{unpair}(e, xy \mapsto y)\rangle \quad \longrightarrow_\eta \quad e$$

5.2 Theories

A first-order *theory* Θ on a given signature and set of predicates is a (possibly infinite) set of closed formulas called *axioms*. A formula A is *provable* in a theory Θ if there is a finite subset $\Gamma \subseteq \Theta$ such that $\Gamma \vdash A$ is provable. Unless otherwise specified, the ambient first order logic is usually taken to be classical when considering first order theories.

5.2.1 Equality. We often consider theories with *equality*. This means that we suppose that we have a predicate "=" of arity 2, together with axioms

$$\forall x.x = x$$
$$\forall x.\forall y.x = y \Rightarrow y = x$$
$$\forall x.\forall y.\forall z.x = y \Rightarrow y = z \Rightarrow x = z$$

and, for every function symbol f of arity n, we have an axiom

$\forall x_1.\forall x_1'.\ldots.\forall x_n.\forall x_n'.$
$$x_1 = x_1' \Rightarrow \ldots \Rightarrow x_n = x_n' \Rightarrow f(x_1,\ldots,x_n) = f(x_1',\ldots,x_n')$$

and, for every predicate P of arity n, we have an axiom

$\forall x_1.\forall x_1'.\ldots.\forall x_n.\forall x_n'.$
$$x_1 = x_1' \Rightarrow \ldots \Rightarrow x_n = x_n' \Rightarrow P(x_1,\ldots,x_n) \Rightarrow P(x_1',\ldots,x_n')$$

These are sometimes called the *congruence axioms*.

Example 5.2.1.1. The *theory of groups* is the theory with equality over the signature $\Sigma = \{\times : 2, 1 : 0\}$ whose axioms are

$$\forall x.1 \times x = x \qquad \forall x.\forall y.\forall z.(x \times y) \times z = x \times (y \times z) \qquad \forall x.\exists y.y \times x = 1$$
$$\forall x.x \times 1 = x \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall x.\exists y.x \times y = 1$$

together with the axioms for equality

$$\forall x.x = x$$
$$\forall x.\forall y.x = y \Rightarrow y = x$$
$$\forall x.\forall y.\forall z.x = y \Rightarrow y = z \Rightarrow x = z$$
$$\forall x.\forall x'.\forall y.\forall y'.x = x' \Rightarrow y = y' \Rightarrow x \times y = x' \times y'$$
$$1 = 1$$

5.2.2 Properties of theories. A theory is

– *consistent* when $\bot$ is not provable in the theory,

– *complete* when for every formula A, either A or $\neg A$ is provable in the theory,

– *decidable* when there is an algorithm which, given a formula A decides whether A is provable in the theory or not.

5.2.3 Models. Theories are thought of as describing structures made of sets and functions satisfying axioms. For instance, the theory of groups of example 5.2.1.1 can be seen as a syntax for groups in the traditional sense. These structures are called *models* of the theory and we very briefly recall those here. We do not even scratch the surface of model theory here, and the reader interested in knowing more about those is urged to read some standard textbooks about that such as [CK90].

Structure. Suppose given a signature Σ and a set $\mathcal{P}$ of predicates. A *structure M* consists of

- a non-empty set M called the *domain* of the structure,

- a function $[\![f]\!] : M^n \to M$ for every function symbol $f \in \Sigma$,

- a relation $[\![P]\!] \subseteq M^n$ for every predicate symbol $P \in \mathcal{P}$.

Interpretation. Suppose fixed such a structure. Given $k \in \mathbb{N}$ and a term t whose free variables are among $\{x_1, \ldots, x_k\}$, we define its *interpretation* as the function

$$[\![t]\!]^k : M^k \to M$$

defined by induction by

$$[\![x_i]\!]^k : M^k \to M$$

is the canonical i-th projection and, for every function symbol f of arity n, and $(m_1, \ldots, m_k) \in M^k$,

$$[\![f(t_1, \ldots, t_n)]\!]^k(m_1, \ldots, m_k) =$$
$$[\![f]\!]([\![t_1]\!]^k(m_1, \ldots, m_k), \ldots, [\![t_n]\!]^k(m_1, \ldots, m_k))$$

where $[\![f]\!]$ is given by the structure and $[\![t_i]\!]^k$ is computed inductively for every index i. In other words, the interpretation of terms is the only extension of the structure which is compatible with composition. Given $k \in \mathbb{N}$ and a formula A whose free variables are among $\{x_1, \ldots, x_k\}$, we define its *interpretation* $[\![A]\!]^k$ as the subset of M^k defined inductively as follows:

$$[\![\bot]\!]^k = \emptyset \qquad\qquad [\![\top]\!]^k = M^k$$
$$[\![A \wedge B]\!]^k = [\![A]\!]^k \cap [\![B]\!]^k \qquad\qquad [\![A \vee B]\!]^k = [\![A]\!]^k \cup [\![B]\!]^k$$
$$[\![\neg A]\!]^k = M^k \setminus [\![A]\!]^k \qquad\qquad [\![A \Rightarrow B]\!]^k = [\![\neg A \vee B]\!]^k$$

together with

$$[\![\forall x_{k+1}.A]\!]^k = \bigcap_{m \in M} \{(m_1, \ldots, m_k) \in M^k \mid (m_1, \ldots, m_k, m) \in [\![A]\!]^{k+1}\}$$

and

$$[\![\exists x_{k+1}.A]\!]^k = \bigcup_{m \in M} \{(m_1, \ldots, m_k) \in M^k \mid (m_1, \ldots, m_k, m) \in [\![A]\!]^{k+1}\}$$

The interpretation of A is thus intuitively the set of values in M for its free variables making it true.

Satisfaction for closed formulas. Given a closed formula A, its interpretation $[\![A]\!]^0$ is a subset of $M^0 = \{()\}$, which is a set with one element, conventionally written (). There are therefore two possible values for $[\![A]\!]^0$: $\emptyset$ and $\{()\}$. In the second case, we say that the formula A is *satisfied* in the structure.

Model. A structure is a *model* of a theory Θ when each formula in Θ is satisfied in the structure.

Example 5.2.3.1. Consider the theory of groups (example 5.2.1.1). A structure consists of

- a set M,

- a function $[\![\times]\!] : M^2 \to M$,

- a constant $[\![1]\!] : M^0 \to M$,

- a relation $[\![=]\!] \subseteq M \times M$.

We say that such a structure has *strict equality* when the interpretation of the equality is the diagonal relation

$$[\![=]\!] = \{(m, m) \mid m \in M\}$$

Such a structure M is a model of the theory of groups, i.e. is a model for all its axioms, precisely if $(M, [\![\times]\!], [\![1]\!])$ is a group in the traditional sense, and conversely every group gives rise to a model where equality is interpreted in such a way: the models with strict equality of the theory of groups are precisely groups.

Remark 5.2.3.2. As can be seen in the previous example, it is often useful to restrict to models with strict equality. Since equality is always a congruence (because of the axioms imposed in section 5.2.1), from any model we can construct a model with strict equality by quotienting the model under the relation interpreting equality, so that this assumption is not very restrictive.

Validity. A sequent
$$y_1 : A_1, \ldots, y_n : A_n \vdash A$$

is *satisfied* in M if for every $k \in \mathbb{N}$ such that the free variables of the sequent are in $\{x_1, \ldots, x_k\}$, we have

$$[\![A_1]\!]^k \cap \ldots \cap [\![A_n]\!]^k \subseteq [\![A]\!]^k$$

which is equivalent to requiring that $[\![\neg(A_1 \wedge \ldots \wedge A_n \Rightarrow A)]\!]^k$ is empty. A sequent is *valid* when it is satisfied in every model.

Correctness. We can now formally state the fact that our notion of semantics is compatible with our logical system.

Theorem 5.2.3.3 (Correctness). *Every derivable sequent is valid.*

Proof. By induction on the derivation of the sequent. $\qquad\square$

The above theorem has the following important particular case:

Corollary 5.2.3.4. For every theory Θ and closed formula A such that $\Theta \vdash A$ is derivable, every model of Θ is also a model of A.

Example 5.2.3.5. In the theory of groups, one can show

$$\forall x.\forall y.\forall y'.x \times y = 1 \Rightarrow y' \times x = 1 \Rightarrow y = y'$$

by formalizing the following sequence of implications of equalities:

$$\begin{aligned}
x \times y &= 1 \\
y' \times (x \times y) &= y' \times 1 \\
y' \times (x \times y) &= y' \\
(y' \times x) \times y &= y' \\
1 \times y &= y' \\
y &= y'
\end{aligned}$$

By correctness, it holds in every group: a left inverse of an element coincides with any right inverse of the same element.

The contrapositive of the above theorem is also quite useful:

Corollary 5.2.3.6. For every theory Θ and closed formula A, if there exists a model of Θ which is not a model of A then $\Theta \vdash A$ is not derivable.

Example 5.2.3.7. In the theory of groups, consider the formula

$$\forall x.\forall y.x \times y = y \times x$$

We know that there exist non-abelian groups, for instance the symmetric group on 3 elements S_3. Such a non-abelian group being a model for the theory of groups but not for the above formula, we can conclude that this formula cannot be deduced in the theory of groups.

Finally, a major consequence of the theorem is the following. A theory is *satisfiable* when it admits a model.

Proposition 5.2.3.8. A satisfiable theory is consistent.

Proof. Suppose that Θ is a theory with a model M. If we had $\Theta \vdash \bot$ then, by corollary 5.2.3.4, we would have that M is a model of $\bot$, which it is not since $[\![\bot]\!] = \emptyset$ by definition. $\square$

Remark 5.2.3.9. As explained in section 5.1.10, the handling of first-order variables in traditional first-order logic is not entirely satisfactory: we do not keep track of the free variables we use. This is why we have to have k (the number of first-order variables) as a parameter for the interpretation. This is also why we need to restrict to non-empty domains in structures. For instance, the sequent $\vdash \exists x.\top$ is always derivable and it would not be satisfied in the structure with an empty domain. See section 5.1.10 for a solution to this issue.

Skolemisation. Suppose fixed a signature Σ and a set $\mathcal{P}$ of predicates. A formula on $(\Sigma, \mathcal{P})$ of the form

$$\forall x.\exists y.A(x,y)$$

states that for every element x there exists a y such that $A(x, y)$ is satisfied. When this formula admits a model, we can construct a function f which to every x associates one of the associated y. Thus it implies that the formula

$$\forall x. A(x, f(x))$$

on $(\Sigma', \mathcal{P})$ is also satisfiable, where the signature Σ' is Σ extended with a symbol f of arity one. By a similar reasoning, one shows that the satisfiability of the second formula implies the satisfiability of the first formula. The two formulas are thus *equisatisfiable*: one is satisfiable if and only if the second is. This process of "replacing existential quantifications by function symbols" is due to Skolem: it allows to replace a theory by another equisatisfiable theory whose axioms do not contain existential quantifications, see section 5.4.6.

More generally, given a formula on $(\Sigma, \mathcal{P})$ of the form

$$\forall x_1. \ldots \forall x_n. \exists y. A$$

a *skolemization* of it is the formula

$$\forall x_1. \ldots \forall x_n. A[f(y_1, \ldots, y_m)/y]$$

on the signature $(\Sigma', \mathcal{P})$ where $\mathrm{FV}(\exists y. A) = \{y_1, \ldots, y_m\}$ and Σ' is Σ extended with a fresh symbol f of arity m.

Proposition 5.2.3.10. A formula of the form $\forall x_1. \ldots \forall x_n. \exists y. A$ is satisfiable if and only if its skolemization is.

Example 5.2.3.11. In the theory of groups from example 5.2.1.1, we can skolemize the axiom $\forall x. \exists y. y \times x = 1$. This forces us to introduce a new unary function symbol i (which will be the function that to an element associates its inverse) and reformulate the axiom as $\forall x. i(x) \times x = 1$.

Remark 5.2.3.12. If we allowed to perform this process for any existential quantification in a formula, proposition 5.2.3.10 would not be true. For instance, the formula $\neg(\exists x. g(x) = x)$ is satisfiable when g has no fixpoint. If we "skolemize" it, we obtain the formula $\neg(g(f()) = f())$ where f is a fresh nullary function symbol: this formula is satisfiable when there is an element which is not a fixpoint for g. The two are thus not equisatisfiable.

5.2.4 Presburger arithmetic. The *Presburger arithmetic* axiomatizes the addition over natural numbers. It is the theory with equality over the signature $\Sigma = \{0 : 0, S : 1, + : 2\}$ whose axioms are those for equality together with

$$\forall x. 0 = S(x) \Rightarrow \bot$$
$$\forall x. \forall y. S(x) = S(y) \Rightarrow x = y$$
$$\forall x. 0 + x = x$$
$$\forall x. \forall y. S(x) + y = S(x + y)$$

together with, for every formula $A(x)$ with one free variable x, an axiom

$$A(0) \Rightarrow (\forall x. A(x) \Rightarrow A(S(x))) \Rightarrow \forall x. A(x)$$

such an infinite family of axioms is sometimes called an *axiom scheme*: it expresses here the *induction* principle.

Example 5.2.4.1. For instance, $\forall x.x + 0 = x$ can be proved by induction on x. Namely, consider the formula $A(x)$ being $x + 0 = x$. We have

– $A(0)$: $0 + 0 = 0$.

– Suppose $A(x)$, we have $A(S(x))$, namely $S(x) + 0 = S(x + 0) = S(x)$.

This theory was shown by Presburger to be consistent, complete and decidable [Pre29]. In the worst case, any decision algorithm has a complexity $O(2^{2^{cn}})$ with respect to the size n of the formula to decide [FR98], although it is useful in practice (it is for example implemented in the tactic `omega` of Coq). It is also very weak: for instance, one cannot define the multiplication function in it (if we could, it would not be decidable, see next section).

5.2.5 Peano and Heyting arithmetic. The *Peano arithmetic*, often written *PA*, extends Presburger arithmetic by also axiomatizing multiplication. It is the theory with equality on the signature $\Sigma = \{0 : 0, S : 1, + : 2, \times : 2\}$ whose axioms are those of equality, those of Presburger arithmetic, and

$$\forall x.0 \times x = 0$$
$$\forall x.\forall y.S(x) \times y = y + (x \times y)$$

This theory is implicitly understood with an ambient classical first order logic. When the logic is intuitionistic, the theory is called *Heyting arithmetic* (or *HA*).

Exercise 5.2.5.1. In HA, prove $\forall x.x + 0 = x$.

Consistency. The second of Hilbert's list of 23 problems posed in 1900 consisted in showing that Peano arithmetic is consistent, i.e. cannot be used to prove $\bot$, or equivalently that $0 = S(0)$ cannot be proved. A natural reaction would be to use corollary 5.2.3.4 and build a model for this theory, whose existence would imply its consistency, and there is an obvious model: the set $\mathbb{N}$ of natural numbers with usual zero, successor, addition and multiplication functions. However, the usual construction of this set of natural numbers is itself performed inside (models of) set theory (see section 5.3) which is a much stronger theory. All this would prove is that if set theory is consistent then Peano arithmetic is consistent, which is like proving that if we have a nuclear bomb then we can kill a fly. This is why people first hoped to prove the consistency of Peano arithmetic in theories as weak as possible and, why not, in Peano arithmetic itself. However, in 1931, Gödel showed in his *second incompleteness theorem* that Peano arithmetic cannot prove its own consistency [Göd31] (unless it is inconsistent). The cut elimination procedure was then introduced by Gentzen in 1936 precisely in order to show consistency of Heyting arithmetic using methods similar to those of theorem 5.1.9.2, although the proof is more involved due to the presence of the axioms of the theory, from which one can deduce the consistency of Peano arithmetic using double negation translations of Peano arithmetic into Heyting arithmetic as in section 2.5.9, see [Gen36].

Induction up to ε_0. Gentzen's proof brings no contradiction with Gödel's theorem, because this proof (or more precisely the proof that the cut-elimination procedure terminates) requires more than the induction principle: we need a

```
(** Finite rooted trees. *)
type tree = T of tree list

(** Lexicographic extension of an order. *)
let rec lex le l1 l2 =
  match l1, l2 with
  | x::l1, y::l2 ->
    if x = y then lex le l1 l2
    else le x y
  | [], _ -> true
  | _, [] -> false

(** Order on trees. *)
let rec le t1 t2 =
  match t1, t2 with
  | T l1, T l2 ->
    let cmp t1 t2 =
      if t1 = t2 then 0
      else if le t1 t2 then -1 else 1
    in
    let l1 = List.sort cmp l1 in
    let l2 = List.sort cmp l2 in
    lex le l1 l2
```

Figure 5.1: ε_0 in OCaml.

transfinite induction up to the ordinal ε_0 (which is ω to the power ω to the power ω and so on, i.e. $\varepsilon_0 = \omega^{\varepsilon_0}$). In other words, while induction only requires us to believe that the set of natural numbers is well-founded, the transfinite induction up to ε_0 now requires that the following set of trees is well-founded. By a classical result in ordinal arithmetic (which we cannot detail here), any ordinal $\alpha < \varepsilon_0$ can be uniquely written as $\alpha = \omega^{\beta_1} + \ldots + \omega^{\beta_n}$ where $\alpha > \beta_1 \geqslant \ldots \geqslant \beta_n$ are ordinals, this is called the Cantor normal form of α, each of the β_i having a similar normal form. Such an ordinal α can thus be represented as a planar rooted tree, with one root and n sons, which are the trees corresponding to the β_i. For instance, the ordinals $\omega^{\omega^3+1} + 2$ and $\omega^{w^3} \cdot 3 + 2$ respectively correspond to the trees
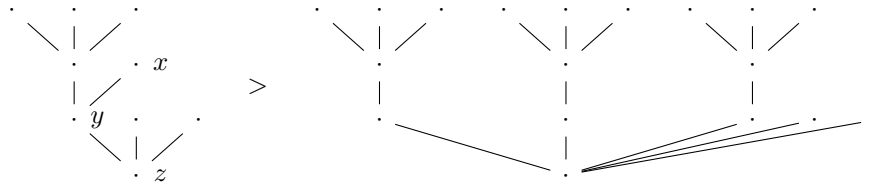
These trees can be compared by lexicographically comparing the sons of the root (which are supposed to be ordered decreasingly), so that for instance, the tree on the left above is greater than the one on the right. An implementation of this order is provided in figure 5.1. This order can also be interpreted using

the following *Hydra game* on trees [KP82]. This game with two players starts with a tree as above and at each turn

- the first player removes a leaf x (a node without sons) of the tree,

- the second player chooses a number n, looks for the parent y of x and the parent z of y (it does nothing if no such parents exist), and adds n copies of the tree with y as root as new children of z.

The game stops when the tree is reduced to its root. We now see where the game draws its name from: the first player cuts the head of the Hydra, but in response the Hydra grows many new heads! For instance, in the figure above, the tree on the right is obtained from the one of the left after one round. Given trees α and β, it can be shown that $\alpha \geqslant \beta$ if and only if β can be obtained after some finite number rounds of the game starting from α. Believing that ε_0 is well-founded is thus equivalent to believing that every such game will necessarily end (try it, to convince yourself that it always does!).

Undecidability. Finally, we would like to mention that Peano arithmetic is also undecidable, which was shown by Turing [Tur37b]. Namely, a sequence of configurations of a Turing machine can be suitably encoded as an integer, so that one can write a formula expressing that a given natural number encodes such a sequence, which ends on an accepting configuration. From there, it is easy to construct a formula expressing the fact that the machine is not halting, and such formulas cannot be decided, otherwise we would decide the halting problem.

5.3 Set theory

Set theory is a first-order theory whose intended models are sets. Everything is a set there, in particular the elements of sets are themselves sets. This theory was defined at the beginning of the 20th century while looking for axiomatic foundations of mathematics. We only briefly scratch the subject and refer to standard textbooks [Kri98, Deh17] for more details.

5.3.1 Naive set theory. The *naive set theory* is the theory with a binary predicate "$\in$" and the following axiom scheme, called *unrestricted comprehension*

$$\exists y. \forall x. x \in y \Leftrightarrow A$$

for every formula A with x as only free variable. Informally, this states for every property $A(x)$, the existence of a set

$$y = \{x \mid A(x)\}$$

of elements x satisfying the property $A(x)$. This theory is surprisingly simple and works surprisingly well: we can perform all the usual constructions:

- the empty set is $\{x \mid \bot\}$,

- the union of two sets is $x \cup y = \{z \mid z \in x \vee z \in y\}$,

- the intersection of two sets is $x \cap y = \{z \mid z \in x \wedge z \in y\}$,

  - the product of two sets is $x \times y = \{(i,j) \mid i \in x \land j \in y\}$ with the notation $(i,j) = \{\{i\}, \{i,j\}\}$,

  - the inclusion of two sets is $x \subseteq y = \forall z.z \in x \Rightarrow z \in y$,

  - the powerset is $\mathcal{P}(x) = \{y \mid y \subseteq x\}$,

and so on.

Russell's paradox. There is only a "slight" problem with this theory: it is inconsistent, meaning that we can in fact prove any formula, which explains why everything was so simple. This was first formalized by Russell in 1901, using what is known nowadays as the *Russell paradox*, which goes as follows. Consider the property

$$A = \neg(x \in x)$$

The unrestricted comprehension scheme ensures the existence of a set y such that

$$\forall x.x \in y \Leftrightarrow \neg(x \in x)$$

In particular, for x being y, we have

$$y \in y \Leftrightarrow \neg(y \in y)$$

In classical logic, we can easily conclude to an inconsistency:

  - if $y \in y$ then $\neg(y \in y)$ and therefore we can prove $\bot$,

  - if $\neg(y \in y)$ then $y \in y$ and therefore we can prove $\bot$.

Russell's paradox in intuitionistic logic. This proof can be thought of as reasoning by case analysis on whether $y \in y$ is true or not and, as such it seems that it is not intuitionistically valid because we are using the excluded middle. However, it can also be considered as a valid intuitionistic proof: namely, the two cases amount to

  - prove $\neg(y \in y)$, and

  - prove $\neg\neg(y \in y)$,

from which we can deduce $\bot$. More generally, for any formula A, one can show intuitionistically that

$$(A \Leftrightarrow \neg A) \Rightarrow \bot$$

Namely, the equivalent formula

$$(A \Rightarrow \neg A) \Rightarrow (\neg A \Rightarrow A) \Rightarrow \bot$$

can be proved by

$$
\cfrac{
  \cfrac{
    \cfrac{\overline{\Gamma, \neg A \vdash \neg A} \ (\text{ax})}{\Gamma, \neg A \vdash \bot} \quad
    \cfrac{\cfrac{\overline{\Gamma, \neg A \vdash \neg A \Rightarrow A} \ (\text{ax}) \quad \overline{\Gamma, \neg A \vdash \neg A} \ (\text{ax})}{\Gamma, \neg A \vdash A} (\Rightarrow_{\text{E}})}{}
  }{\cfrac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash \neg\neg A} (\neg_{\text{I}})} (\neg_{\text{E}})
  \qquad
  \cfrac{
    \cfrac{\overline{\Gamma, A \vdash A \Rightarrow \neg A} \ (\text{ax}) \quad \overline{\Gamma, A \vdash A} \ (\text{ax})}{\cfrac{\Gamma, A \vdash \neg A}{\cfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} (\neg_{\text{I}})} (\Rightarrow_{\text{E}}) \quad \overline{\Gamma, A \vdash A} \ (\text{ax})} (\Rightarrow_{\text{E}})
  }{}
}{\cfrac{\Gamma \vdash \bot}{\vdash (A \Rightarrow \neg A) \Rightarrow (\neg A \Rightarrow A) \Rightarrow \bot} (\Rightarrow_{\text{I}})} (\neg_{\text{E}})
$$

with $\Gamma = A \Rightarrow \neg A, \neg A \Rightarrow A$, whose corresponding proof-term is

$$\lambda f^{A \Rightarrow \neg A}.\lambda g^{\neg A \Rightarrow A}.(\lambda x^{\neg A}.x\,(g\,x))(\lambda a^{A}.f\,a\,a)$$

Another, more symmetrical proof term for the same formula is

$$\lambda f^{A \Rightarrow \neg A}.\lambda g^{\neg A \Rightarrow A}.f\,(g\,(\lambda a^{A}.f\,a\,a))\,(g\,(\lambda a^{A}.f\,a\,a))$$

In both cases, note that we recover the looping term $\Omega = (\lambda x.xx)(\lambda x.xx)$ if we apply it to the identity twice.

The so-called *Curry paradox* is the following slight generalization of the above formula

$$(A \Leftrightarrow (A \Rightarrow B)) \Rightarrow B$$

and can be shown using the same λ-terms.

Size issues. The problem with naive set theory is due to size: the collection of all sets is "too big" to actually form a set. Once this issue was identified, subsequent attempts at formalizing set theory have struggled to take it in account. We should not be able to consider this as a set and therefore we cannot consider the set of all sets which satisfy a property, such as not belonging to itself...

Other paradoxes. Other paradoxes can be used to show the inconsistency of naive set theory. For instance, an argument based on Cantor's theorem is the following one. Suppose that there exists a set u of all sets. Every subset x of u is a set, and thus an element of u. In this way, we can construct an injection from the powerset of u to u, which is excluded by Cantor's diagonal argument, see appendix A.4. Another classical paradox is the one of Burali-Forti, presented in section 8.2.3.

5.3.2 Zermelo-Fraenkel set theory. The above observations lead to a refined axiomatic for set theory, the most popular being called *Zermelo-Fraenkel set theory*, or *ZF* [Zer08]. We make a very brief presentation of it here, mostly discussing the axiom of choice. This is the classical first order theory with equality with a binary predicate $\in$, whose axioms are the following.

Axiom of extensionality. This axiom states that two sets with the same elements are equal:

$$\forall x.\forall y.((\forall z.z \in x \Leftrightarrow z \in y) \Rightarrow x = y)$$

If we introduce the notation $x \subseteq y$ for the formula $\forall z.z \in x \Rightarrow z \in y$ which expresses that the set x is included in the set y, the axiom of extensionality can be rephrased as

$$\forall x.\forall y.(x \subseteq y \wedge y \subseteq x) \Rightarrow x = y$$

i.e. two sets are equal precisely when they have the same elements.

Axiom of union. This axiom states that the union of the elements of a set is still a set:

$$\forall x.\exists y.\forall i.(i \in y \Leftrightarrow \exists z.(i \in z \wedge z \in x))$$

In more usual notation, this states the existence, for every set x, of the set

$$y = \bigcup x = \bigcup_{z \in x} z$$

In particular, we can construct the union of two sets x and y as

$$x \cup y = \bigcup \{x, y\}$$

where the set $\{x, y\}$ is constructed using the axiom schema of replacement, see below.

Axiom of powerset. This axiom states that given a set x, there is a set whose elements are precisely the subsets of x, usually called the *powerset* of x and written $\mathcal{P}(x)$:

$$\forall x.\exists y.\forall z.(z \in y \Leftrightarrow (\forall i.i \in z \Rightarrow i \in x))$$

In more usual notation,

$$\forall x.\exists y.\forall z.(z \in y \Leftrightarrow z \subseteq x)$$

i.e. we can construct the set

$$y = \mathcal{P}(x) = \{z \mid z \subseteq x\}$$

Axiom of infinity. The axiom of infinity states the existence of an infinite set:

$$\exists x.(\emptyset \in x \wedge \forall y.y \in x \Rightarrow S(y) \in x)$$

where the empty set $\emptyset$ is defined using the axiom schema of replacement below and $S(y) = y \cup \{y\}$ is the *successor* of a set. A set is called *inductive* when it contains the empty set and is closed under successor: the axiom states the existence of an inductive set. In particular, the set $\mathbb{N}$ of natural numbers can be constructed as the intersection of all inductive sets. Here, the natural numbers are encoded following the von Neumann convention:

$$0 = \emptyset \qquad 1 = 0 \cup \{0\} = \{\emptyset\} \qquad 2 = 1 \cup \{1\} = \{\emptyset, \{\emptyset\}\} \qquad \dots$$

and more generally $n + 1 = n \cup \{n\}$. The definition implies immediately the following principle of induction: every inductive subset of the natural numbers is the set of natural numbers.

Axiom schema of replacement. This axiom states that the image of a set under a partial function is a set:

$$(\forall i.\forall j.\forall j'.(A \wedge A[j'/j] \Rightarrow j = j')) \Rightarrow \forall x.\exists y.\forall j.(j \in y \Leftrightarrow \exists i.(i \in x \wedge A))$$

where A is any formula such that $j' \notin \mathrm{FV}(A)$ (but A might contain i or j or other free variables). This is thus an axiom schema: it is an infinite family of axioms, one for each such formula A.

For simplicity, we consider the case where the formula contains only i and j as free variables, and is thus written $A(i, j)$. In this case the axiom reads as

$$(\forall i.\forall j.\forall j'.(A(i,j) \wedge A(i,j') \Rightarrow j = j')) \Rightarrow$$
$$\forall x.\exists y.\forall j.(j \in y \Leftrightarrow \exists i.(i \in x \wedge A(i,j)))$$

The formula A encodes a relation: a set i is in relation with a set j when $A(i,j)$ is true. In particular, the relation corresponds to a partial function when every element i is in relation with at most one element j, i.e.

$$\forall i.\forall j.\forall j'.(A(i,j) \wedge A(i,j') \Rightarrow j = j')$$

Namely, such a relation corresponds to the partial function f from sets to sets such that $f(i)$ is the unique j such that $A(i,j)$, should there exists one, and is undefined otherwise. Then, our axiom states that given a set x, we can construct the set

$$y = \{j \mid \exists i \in x.A(i,j)\}$$

of its images under f.

For instance, the empty set $\emptyset$ is defined as the set y obtained in this way from any set x (and there exists one by the axiom of infinity) using the nowhere defined function, which can be encoded as the relation $A = \bot$:

$$\emptyset = \{j \mid \exists i \in x.\bot\}$$

Given two sets x and y, we can construct the set $\{x, y\}$ as the image of the partial function over the natural numbers sending 0 (i.e. $\emptyset$) to x and 1 (i.e. $\{\emptyset\}$) to y:

$$\{x, y\} = \{j \mid \exists i \in \mathbb{N}.(i = 0 \wedge j = x) \vee (i = 1 \wedge j = y)\}$$

and we can similarly construct a set containing any finite given family of sets. Given two sets x and y, we can construct their intersection as

$$x \cap y = \{j \mid \exists i \in x \cup y.j = i \wedge i \in x \wedge i \in y\}$$

Given two sets x and y, we can encode a pair of elements $i_1 \in x$ and $i_2 \in y$ as $(i_1, i_2) = \{\{i_1\}, \{i_1, i_2\}\}$, which is an element of $\mathcal{P}(\mathcal{P}(x \cup y))$, and thus construct the product of the two sets as

$$x \times y = \{j \mid \exists i \in \mathcal{P}(\mathcal{P}(x \cup y)).\exists i_1.\exists i_2.j = i \wedge i = (i_1, i_2) \wedge i_1 \in x \wedge i_2 \in y\}$$

More generally, given a predicate $B(i)$ and a set x, we can we can construct the set of elements of x satisfying B as

$$\{i \in x \mid B(i)\} = \{j \mid i = j \wedge B(i)\}$$

This construction corresponds to what is sometimes called the *axiom schema of restricted comprehension* and can formally be stated as

$$\forall x.\exists y.\forall i.(i \in y \Leftrightarrow (i \in x \wedge A))$$

where A is a formula such that $x, y \notin \mathrm{FV}(A)$, but A might contain i or other free variables, i.e. in usual notation, for every set x we can construct the set

$$y = \{i \in x \mid A\}$$

Note that compared to the unrestricted comprehension scheme, which was at the source of Russell's paradox, we can only construct the set of elements <u>of some set</u> which satisfy A.

Axiom of foundation. The axiom of foundation states that every non-empty set contains a member which is disjoint from the whole set:

$$\forall x.(\exists y.y \in x) \Rightarrow \exists y.(y \in x \wedge \neg \exists i.(i \in y \wedge i \in x))$$

or, in modern notation,

$$\forall x.x \neq \emptyset \Rightarrow \exists y \in x.y \cap x = \emptyset$$

One of the main consequences of the axiom of foundation is the following:

Lemma 5.3.2.1. There is no infinite sequence of sets (x_i) such that $x_{i+1} \in x_i$.

Proof. Suppose the contrary. The sequence of sets can be seen as a function f with $\mathbb{N}$ as domain, which to every i associates $f(i) = x_i$. By the axiom schema of replacement, its image $x = \{x_i \mid i \in \mathbb{N}\}$ is also a set and by the axiom of foundation there exists $y \in x$ such that $y \cap x \neq \emptyset$. By definition of x, there exists some natural number i for which $y = f(i) = x_i$. However, we have $x_{i+1} \in x_i$ and therefore $x_{i+1} \in y \cap x$. Contradiction. $\qquad\square$

In particular, there is not set x such that $x \in x$ (otherwise, the constant sequence $x_i = x$ would contradict previous lemma).

The axiom of foundation is, in presence of the other axioms, equivalent to the following, better looking, *axiom of induction*, which is a variant of transfinite induction (sometimes called $\in$-*induction*):

$$(\forall x.(\forall y.y \in x \Rightarrow A(y)) \Rightarrow A(x)) \Rightarrow \forall x.A(x)$$

for every predicate A with $\mathrm{FV}(A) \subseteq \{x\}$.

Avoiding Russell's paradox. Intuitively, the way ZF avoids Russell's paradox is by considering that the collections such as the collection of all sets are "too big" to be sets themselves: they are sometimes called *classes* and a set can be considered as a "small class".

For instance, we have the restricted schema of replacement, but not the unrestricted one, which would allow defining the set of all sets as $\{x \mid \top\}$: we can only consider the collection of elements satisfying some property within a set, i.e. a subset of a small class is itself small. This is also the reason why, in the axiom scheme of replacement, we require $A(x, y)$ to be functional: otherwise, for a given x the set $\{y \mid A(x, y)\}$ would not guaranteed to be a set (it could be too big). Also, we have seen that the axioms of foundation ensures that no set contains itself, which avoids that classes, such as the collection of all sets, are themselves sets.

The axiom of choice. The *axiom of choice* states that given a collection x of non-empty sets, we can pick an element in each of the sets:

$$\forall x.\emptyset \notin x \Rightarrow \exists (f : x \to \bigcup x).\forall y \in x.f(y) \in y$$

This states that given a set x of non-empty sets (i.e. x does not contain $\emptyset$), there exists a function f from x to $\bigcup x$ (the union of the elements of x, which can be constructed with the axiom of union) which, for every set $y \in x$ picks an

element of y (i.e. $f(y) \in y$): this is called a *choice function* for x. The careful reader will notice that the existence of a function is not a formal statement of our language but it can be encoded: the formula $\exists (f : x \to y).A$ asserting the existence of a function f from x to y such that A, is a notation for a formula of the form

$$\exists f.f \subseteq x \times y \wedge \ldots$$

which would state (details left to the reader) the existence of a subset f of $x \times y$ which, as a relation, encodes a total function such that A is satisfied.

The axiom of choice has a number of classically equivalent formulations among which

– every relation defined everywhere contains a function,

– every surjective function admits a section,

– the product of a family of non-empty sets is non-empty,

– every set can be well-ordered,

and so on.

5.3.3 Intuitionistic set theory. Set theory, as any other theory can also be considered within intuitionistic first order logic, in which case it is called *IZF*. The reason for is the usual one: we want to be able to exhibit explicit witnesses when constructing elements of sets. We will however see that there is a price to pay for this, which is that things behave much differently than usual: intuitionism is not necessarily intuitive, see [Bau17] for a very good general introduction to the subject.

Equivalent formulations of excluded middle. Most of the proofs related to the excluded middle in set theory are using the following simple observation. Given a proposition A, which might contain any free variable except y, consider the set

$$x = \{y \in \mathbb{N} \mid A\}$$

Then, given a natural number $y \in \mathbb{N}$, we have $y \in x$ if and only if A holds:

$$(y \in x) \Leftrightarrow A \tag{5.1}$$

(in practice, we often use $y = 0$ as arbitrary natural number). In particular, when $A = \bot$, the set x is (by definition) the empty set $\emptyset$ and we have $y \in \emptyset$ if and only if $\bot$. By the axiom of extensionality, we thus have that $x = \emptyset$ is equivalent to $\forall y \in \mathbb{N}.(y \in x) \Leftrightarrow (y \in \emptyset)$ which is equivalent to $A \Leftrightarrow \bot$, which is equivalent to $A \Rightarrow \bot$ (since $\bot \Rightarrow A$ always holds): we have shown

$$(x = \emptyset) \Leftrightarrow \neg A \tag{5.2}$$

For instance, a typical thing we cannot do in IZF is test whether an element belongs to a given set or not:

Lemma 5.3.3.1. In IZF, the formula

$$\forall y. \forall x. (y \in x) \vee (y \notin x)$$

is satisfied if and only if the law of excluded middle is.

Proof. The right-to-left implication is obvious. For the left-to-right implication, given any formula A, consider the natural number $y = 0$ and the set $x = \{y \in \mathbb{N} \mid A\}$: the set x contains the element 0 if and only if A holds. We conclude using (5.1): the above proposition would imply

$$(0 \in \{x \in \mathbb{N} \mid A\}) \vee (0 \notin \{x \in \mathbb{N} \mid A\})$$

which is equivalent to

$$A \vee \neg A$$

and we conclude. $\square$

The intuition behind this result is the following one. In a constructive world, an element of $x = \{y \in \mathbb{N} \mid A\}$ consists of as an element of $\mathbb{N}$ together with a proof that A holds. Therefore, in order to decide whether 0 belongs to x or not, we have to decide whether A holds or not.

Considering the variant of the excluded middle recalled in lemma 2.3.5.3, similarly, we cannot test a set for emptiness either:

Lemma 5.3.3.2. In IZF, the formula

$$\forall x.(x = \emptyset) \vee (x \neq \emptyset)$$

is satisfied if and only if we can prove

$$\neg A \vee \neg\neg A$$

for every formula A.

Proof. The right-to-left implication is clear. For the left-to-right implication, given a formula A, consider the set $x = \{y \in \mathbb{N} \mid A\}$. We conclude using (5.2): we have $x = \emptyset$ if and only if $0 \in \{y \in \mathbb{N} \mid A\} \Leftrightarrow 0 \in \{y \in \mathbb{N} \mid \bot\}$, if and only if $A \Leftrightarrow \bot$, if and only if $\neg A$. $\square$

More generally, we do not expect to be able to decide equality either: the formula

$$\forall x.\forall y.(x = y) \vee (x \neq y)$$

would imply that we can test for emptiness as a particular case. Of course, this does not imply that we cannot decide the equality of some particular sets. For instance, one can show that $0 = \emptyset \neq \{\emptyset\} = 1$ (because $\emptyset$ belongs to 1 but not to 0) and therefore, writing

$$\mathbb{B} = \{0, 1\} = \{x \in \mathbb{N} \mid x = 0 \vee x = 1\}$$

for the set of booleans, we can decide the equality of booleans. By a similar reasoning, we can decide the equality of natural numbers.

Many other "unexpected" properties of IZF (compared to the classical case) can be proved along similar lines. For instance, the finiteness of subsets of a finite set is equivalent to being classical. By a *finite* set, we mean here a set x for which there is a natural number n and a bijection $f : \{0, \ldots, n-1\} \to x$.

Lemma 5.3.3.3. In IZF, every subset of a finite set is finite if and only if the law of excluded middle is satisfied.

Proof. Suppose that every finite subset of a finite set is finite. Given a property A, consider the set $x = \{y \in \mathbb{B} \mid A\}$, which is a subset of the finite set $\mathbb{B}$ of booleans. By hypothesis, this set is finite and therefore there exists a natural number n and a function f as above. Since we can decide equality for natural numbers as argued above, we have either $n = 0$ or $n \neq 0$: in the first case $x = \emptyset$ and thus $\neg A$ holds, in the second case, $f(0) \in x$ and thus A holds. We therefore have $A \vee \neg A$. Conversely, in classical logic, every subset of a finite set is finite, as everybody knows. $\qquad\square$

The axiom of choice. Seen from a constructive perspective the axiom of choice is quite dubious: it allows the construction of an element in each set of a family of non-empty sets, without having to provide any hint at how such an element could be constructed. In particular, given a non-empty set x, the axiom of choice provides a function $f : \{x\} \to x$, i.e. an element of x (the image of x under f), and allows proving

$$x \neq \emptyset \Rightarrow \exists y.y \in x$$

i.e. we can construct an element in x by only knowing that there exists one. This is precisely the kind of behavior we invoked in section 2.5.2 in order to motivate the fact that double negation elimination was not constructive. In fact, we will see below that having the axiom of choice implies that the ambient logic is classical.

Another reason why the axiom of choice can be questioned is that it allows proving quite counter-intuitive results, the most famous perhaps being the Banach-Tarski theorem recalled below. Two sets A and B of points in $\mathbb{R}^3$ are *congruent* if one can be obtained from the other by an isometry, i.e. by using translations, rotations and reflections.

Theorem 5.3.3.4 (Banach-Tarski). Given two bounded subsets of $\mathbb{R}^3$ of non-empty interior, there are partitions

$$A = A_1 \uplus \ldots \uplus A_n \qquad\qquad B = B_1 \uplus \ldots \uplus B_n$$

such that A_i is congruent to B_i for $1 \leqslant i \leqslant n$.

Proof. Using the axiom of choice and other ingredients... $\qquad\square$

In particular, consider the case where A is a ball in $\mathbb{R}^3$ and B is two copies of the ball A. The theorem states that there is a way to partition the ball A and move the subsets of the partition using isometries only, in order to make two balls. If you try this at home, you should convince yourself that there is no easy way to do so.

For such reasons, people started to investigate the status of the axiom of choice with respect to ZF. In 1938, Gödel constructed a model of ZFC (i.e. a model of ZF satisfying the axiom of choice) inside an arbitrary model of ZF, thus showing that ZFC is consistent if ZF is [Göd38]. In 1963, Cohen showed that the situation is similar with the negation of the axiom of choice. The axiom of choice is thus *independent* of ZF: neither this axiom nor its negation is a consequence of the axioms of ZF and one can add it or its negation without affecting consistency.

Constructivists however will reject the axiom of choice, because it implies the excluded middle:

Theorem 5.3.3.5. In IZF with the axiom of choice, the law of elimination of double negation holds.

Proof. Fix a formula A and suppose $\neg\neg A$ holds. The set

$$x = \{y \in \mathbb{N} \mid A\}$$

is not empty. Namely, we have seen in (5.2) that $x = \emptyset$ implies $\neg A$ which, together with the hypothesis $\neg\neg A$, implies $\bot$. By the axiom of choice, the fact that $x \neq \emptyset$ implies the existence of an element of x because we have a choice function for $\{x\}$, which implies A by (5.1). Therefore $\neg\neg A \Rightarrow A$. $\square$

The above proof is not entirely satisfactory because it uses the following form of the axiom of choice:

> any set y, whose elements x are not empty, admits a choice function.

The issue here is that we suppose that an element x of y is not empty, i.e. it is not the case that x does not contain an element. From a constructive point of view, this is not equivalent to supposing that it contains an element (not not containing an element does not mean that we contain an element, because we do not admit double negation elimination) and the latter is more constructive. A better formulation of the axiom of choice would thus be

> any set y, whose elements x contain an element, admits a choice function.

This hints at the fact that we should be careful in IZF about what we mean by the axiom of choice: formulations which were equivalent in classical logic are not any more in intuitionistic logic. This second formulation of the axiom of choice still implies the excluded middle as first noticed by Diaconescu [Dia75, GM78], but this is much more subtle:

Theorem 5.3.3.6 (Diaconescu). In IZF with the axiom of choice, the law of excluded middle is necessarily satisfied.

Proof. Fix an arbitrary formula A: we are going to show $\neg A \vee A$. Consider the sets

$$x = \{z \in \mathbb{B} \mid (z = 0) \vee A\} \qquad \text{and} \qquad y = \{z \in \mathbb{B} \mid (z = 1) \vee A\}$$

Those sets are not empty since $0 \in x$ and $1 \in y$. By the axiom of choice, there is therefore a function $f : \{x, y\} \to \mathbb{B}$ such that $f(x) \in x$ and $f(y) \in y$. Now, $f(x)$ and $f(y)$ are booleans, where equality is decidable, so that we can reason by case analysis on those.

– If $f(x) = f(y) = 0$ then $0 \in y$ thus $(0 = 1) \vee A$ holds, thus A holds.

– If $f(x) = f(y) = 1$ then $1 \in x$ thus $(1 = 0) \vee A$ holds, thus A holds.

– If $f(x) = 0 \neq 1 = f(y)$ then $x \neq y$ (otherwise, $f(x) = f(y)$ would hold), and we have $\neg A$: namely, supposing A, we have $x = y = A$, and thus $\bot$ since $x \neq y$,

– If $f(x) = 1 \neq 0 = f(y)$ then we can show both A and $\neg A$ as above (so that this case cannot happen).

Therefore, we have $\neg A \vee A$. □

This motivates, for the reader interested in intuitionistic logic, which we hope you are by now, the exploration of set theory without choice, but you should be warned that this theory behaves much differently than usual. For instance, Blass has shown the following result [Bla84]:

Theorem 5.3.3.7. In ZF, the axiom of choice is equivalent to the fact that every vector space has a basis.

In fact, we know models of ZF where there is a vector space admitting no basis, and one admitting two basis of different cardinalities.

Synthetic differential geometry. Since classical logic is obtained by adding axioms (e.g. excluded middle) to intuitionistic logic, a proof in intuitionistic logic is valid in classical logic (we are not using the extra axioms). Therefore, one is tempted to think that intuitionistic logic is less powerful than classical logic, because it can prove less. Well, this is true, but this can also be seen as a strength: this also means that we have more models of intuitionistic theories than their classical counterparts. We would like to give an illustration of this.

The notion of *infinitesimal* is notoriously difficult to define in analysis. Intuitively, such a quantity is so small that it should be "almost 0"; in particular, it should be smaller that any usual strictly positive real number. Having such a notion is quite useful. For instance, we expect the derivative of a function $f : \mathbb{R} \to \mathbb{R}$ at x be defined as

$$f'(x) = (f(x + \varepsilon) - f(x))/\varepsilon$$

for any non-zero infinitesimal ε. Namely, $f'(x)$ should be the slope of the line tangent to the slope of f at x, i.e.

$$f(x + \varepsilon) = f(x) + f'(x)\varepsilon$$

More precisely, by "almost 0", we mean here that it should capture first-order variations, i.e. it should be so small that $\varepsilon^2 = 0$. If we are ready to accept the existence of such entities, we find out that computations which traditionally involve subtle concepts such as limits, become simple algebraic manipulations. For instance, consider the function $f(x) = x^2$. We have

$$f(x + \varepsilon) = (x + \varepsilon)^2 = x^2 + 2x\varepsilon + \varepsilon^2 = x^2 + 2x\varepsilon$$

and therefore we should have $f'(x) = 2x$, as expected.

This suggests that we define the set of infinitesimals as

$$D = \{\varepsilon \in \mathbb{R} \mid \varepsilon^2 = 0\}$$

and postulate the following *principle of microaffineness*:

Axiom 5.3.3.8. Every function $f : D \to \mathbb{R}$ is of the form

$$f(\varepsilon) = a + b\varepsilon$$

for some unique reals a and b.

Once this axiom postulated, we necessarily have $a = f(0)$ and we can define $f'(x)$ to be the coefficient b. We have already given an example of such a computation above. We can similarly, compute the derivative of a product of two functions by

$$
\begin{aligned}
(f \times g)'(x + \varepsilon) \\
&= f(x + \varepsilon) \times g(x + \varepsilon) \\
&= (f(x) + f'(x)\varepsilon) \times (g(x) + g'(x)\varepsilon) \\
&= (f(x) + g(x)) + (f'(x)g(x) + f(x)g'(x))\varepsilon + (f'(x) + g'(x))\varepsilon^2 \\
&= (f(x) + g(x)) + (f'(x)g(x) + f(x)g'(x))\varepsilon
\end{aligned}
$$

and therefore $(f \times g)'(x) = f'(x)g(x) + f(x)g'(x)$ as expected. Similarly, the derivative of the composite of two functions can be computed by

$$
g(f(x + \varepsilon)) = g(f(x) + f'(x)\varepsilon) = g(f(x)) + g'(f(x))f'(x)\varepsilon
$$

because $f'(x)\varepsilon$ is easily shown to be an infinitesimal and therefore

$$
(g \circ f)'(x) = g'(f(x))f'(x)
$$

This is wonderful, except that our microaffineness seems to be clearly wrong. Namely,

$$
\varepsilon^2 = 0 \qquad \text{implies} \qquad \varepsilon = 0
$$

thus $D = \{0\}$, and therefore any coefficient b would suit. However... the above implication uses classical reasoning. Namely: if $\varepsilon \neq 0$, we have

$$
\varepsilon = \varepsilon^2/\varepsilon = 0/\varepsilon = 0
$$

from which we can conclude that $\varepsilon = 0$... in classical logic! In intuitionistic logic, all that we have proved is that

$$
\neg\neg(\varepsilon = 0)
$$

This is the sense in which ε is infinitesimal: it is not nonzero.

This shows that there is no obvious contradiction in our axiomatic if we work in intuitionistic logic, but it does not prove that there is no contradiction. This can however be done by constructing models. The field of *synthetic differential geometry* takes this idea of working in intuitionistic logic in order to define infinitesimals as a starting point to study differential geometry [Bel98, Koc06].

5.4 Unification

Suppose fixed a signature. Given two terms t and u, a very natural question is: is there a way to substitute their variables in order to make them equal? In other words, we are trying to solve the equation

$$
t = u
$$

One quickly finds out that there is quite often an infinite number of solutions, and we refine the question to: is there a "smallest" way of substituting the

variables of t and u in order to make them equal? Occurrences of this problem have for instance already been encountered in section 4.4. We explain here how to properly formulate the problem, which we have already encountered in section 4.4.2, and exhibit an algorithm in order to solve it. A detailed introduction to the subject can be found in [BN99].

5.4.1 Equation systems. An *equation* is a pair of terms (t, u) often written

$$t \overset{?}{=} u$$

where t and u are respectively called the *left* and *right member* of the equation. A substitution σ, see section 5.1.3, is a *solution* of the equation when

$$t[\sigma] = u[\sigma]$$

in which case we also say that σ is an *unifier* of t and u. An *equation system*, or *unification problem*, E is a finite set of equations. A substitution σ is a *solution* (or an *unifier*) of E when it is a solution of every equation in E. We write $E[\sigma]$ for the equation system obtained by applying a substitution σ to every member of an equation of E: σ is thus a solution of E when all the equations of $E[\sigma]$ are of the form $t \overset{?}{=} t$.

Example 5.4.1.1. Let us give some examples of unifiers. We suppose that our signature comprises two binary function symbols f and g, and two nullary symbols a and b.

- $f(x, b()) \overset{?}{=} f(a(), y)$ has one unifier: $[a()/x, b()/y]$,

- $x \overset{?}{=} f(y, z)$ has many unifiers: $[f(y, z)/x]$, $[f(a(), z)/x, a()/y]$, etc.

- $f(x, y) \overset{?}{=} g(x, y)$ has no unifier,

- $x \overset{?}{=} f(x, y)$ has no unifier.

Since the solution to an equation system is not unique in general, we can wonder whether there is a best one in some sense when there is one. We will see that it is indeed the case.

5.4.2 Most general unifier. A *preorder* $\leqslant$ is a reflexive and transitive relation: a *partial order* is an antisymmetric preorder. We can define a preorder $\leqslant$ on substitutions by setting $\sigma \leqslant \tau$ whenever there exists a substitution σ' such that $\tau = \sigma' \circ \sigma$.

Example 5.4.2.1. With

$$\sigma = [f(y)/x] \qquad \sigma' = [g(x, x)/y] \qquad \tau = [f(g(x, x))/x, g(x, x)/y]$$

we have $\sigma' \circ \sigma = \tau$ and thus $\sigma \leqslant \tau$.

A *renaming* is a substitution which replaces variables by variables (as opposed to general terms). The relation $\leqslant$ defined above is "almost" a partial order, in the sense that it would be so if we considered substitutions up to renaming:

Lemma 5.4.2.2. Given substitutions σ and τ, we have both $\sigma \leqslant \tau$ and $\tau \leqslant \sigma$ if and only if there exists a renaming σ' such that $\sigma' \circ \sigma = \tau$.

Suppose fixed an equation system E. It easy to see that its set of solutions is upward closed:

Lemma 5.4.2.3. Given substitutions σ and τ such that $\sigma \leqslant \tau$, if σ is a solution of E then τ is also a solution of E.

A solution σ of E is a *most general unifier* when it generates all the solutions by upward closure, i.e. when τ is a solution of E if and only if $\sigma \leqslant \tau$. We will see in next section that when an equation systems admits an unifier, it always admits a most general one, and we have an algorithm to efficiently compute it. We will thus prove, in a constructive way, the following:

Theorem 5.4.2.4. An equation system E has a solution if and only if it has a most general unifier.

5.4.3 The unification algorithm. Suppose given an equation system E, for which we are trying to compute a most general unifier. The idea of the algorithm is to apply a series of transformations to E, which preserve the set of solutions of the system, in order to simplify it and compute a solution. More precisely, our goal is put the equation system in the following form: an equation system E is in *solved form* when

- if is of the form
$$E = \{x_1 \stackrel{?}{=} t_1, \ldots, x_n \stackrel{?}{=} t_n\}$$
  i.e. all its equations have a variable as left member,

- no variable in a left member of an equation occurs in a right member: $x_i \notin \mathrm{FV}(t_j)$ for every indices i and j,

- variables in left members are distinct: $x_i = x_j$ implies $i = j$.

To every equation system in solved form E as above, one can canonically associate the substitution
$$\sigma_E = [t_1/x_1, \ldots, t_n/x_n]$$

Lemma 5.4.3.1. Given an equation system in solved form E, the substitution σ_E is a most general unifier of E.

Given an equation system E, the *unification algorithm*, due to Herbrand [Her30] and Robinson [Rob65], applies the transformations of figure 5.2, in an arbitrary order, and terminates when no transformation applies. We write

$$E \qquad \rightsquigarrow \qquad E'$$

to indicate that the transformation replaces E by E'. At some point of the execution, the algorithm might fail, which we write

$$E \qquad \rightsquigarrow \qquad \bot$$

Example 5.4.3.2. Suppose that the signature comprises symbols a of arity 0, f of arity 3, and g and h of arity 1. Consider the equation system

$$E = \{f(a(), g(x), g(x)) \stackrel{?}{=} f(a(), y, g(h(z)))\}$$

Decompose (we propagate equation to subterms):

$$\{f(t_1, \ldots, t_n) \overset{?}{=} f(u_1, \ldots, u_n)\} \cup E \qquad \rightsquigarrow \qquad \{t_1 \overset{?}{=} u_1, \ldots, t_n \overset{?}{=} u_n\} \cup E$$

Clash (different symbols cannot be unified): for $f \neq g$,

$$\{f(t_1, \ldots, t_n) \overset{?}{=} g(u_1, \ldots, u_m)\} \qquad \rightsquigarrow \qquad \bot$$

Delete (we remove trivial equations):

$$\{f(t_1, \ldots, t_n) \overset{?}{=} f(t_1, \ldots, t_n)\} \cup E \qquad \rightsquigarrow \qquad E$$

Orient (we want variables as left members):

$$\{f(t_1, \ldots, t_n) \overset{?}{=} x\} \cup E \qquad \rightsquigarrow \qquad \{x \overset{?}{=} f(t_1, \ldots, t_n)\} \cup E$$

Occurs-check (we eliminate cyclic equations): when $x \in \mathrm{FV}(t_1) \cup \ldots \cup \mathrm{FV}(t_n)$,

$$\{x \overset{?}{=} f(t_1, \ldots, t_n)\} \qquad \rightsquigarrow \qquad \bot$$

Propagate (we propagate substitutions): when $x \notin \mathrm{FV}(t)$ and $x \in \mathrm{FV}(E)$,

$$\{x \overset{?}{=} t\} \cup E \qquad \rightsquigarrow \qquad \{x \overset{?}{=} t\} \cup E[t/x]$$

Figure 5.2: The unification algorithm.

We have

$$
\begin{aligned}
E &\rightsquigarrow \{a() \not\doteq a(), g(x) \not\doteq y, g(x) \not\doteq g(h(z))\} && \text{by Decompose,}\\
&\rightsquigarrow \{g(x) \not\doteq y, g(x) \not\doteq g(h(z))\} && \text{by Delete,}\\
&\rightsquigarrow \{y \not\doteq g(x), g(x) \not\doteq g(h(z))\} && \text{by Orient,}\\
&\rightsquigarrow \{y \not\doteq g(x), x \not\doteq h(z)\} && \text{by Decompose,}\\
&\rightsquigarrow \{y \not\doteq g(h(z)), x \not\doteq h(z)\} && \text{by Propagate.}
\end{aligned}
$$

The *size* $|t|$ of a term is the number of function symbols occurring in it:

$$
|x| = 0 \qquad\qquad |f(t_1, \ldots, t_n)| = 1 + \sum_{i=1}^{n} |t_i|
$$

Theorem 5.4.3.3. Given any equation system E as input the unification algorithm always terminates. It fails if and only if E has no solution, otherwise the equation system E' at the end of the execution is in solved form and $\sigma_{E'}$ is a most general unifier of E.

Proof. This is detailed in [BN99, section 4.6]. Termination can be shown by observing that the rules make the size of the equation system E decrease: here, the size is the triple (n_1, n_2, n_3) of natural numbers, ordered lexicographically, where n_1 is the number of unsolved variables (a variable is solved when it occurs exactly once in E, as a left member of an equation), $n_2 = \sum_{(t \not\doteq u) \in E} |t| + |u|$ is the size of the equation system and n_3 is the number of equations of the form $t \not\doteq x$ in E. The other properties result from the fact that the transformations preserve the set of unifiers ($\bot$ has no unifier by convention) and that the resulting equation system is in solved form. $\square$

Example 5.4.3.4. The most general unifier of example 5.4.3.2 is

$$
[g(h(z))/y, h(z)/x]
$$

Remark 5.4.3.5. The side conditions of Propagate are quite important (and often forgotten by students when first implementing unification). Without those, unification problems such as $\{x \not\doteq f(x)\}$ would lead to an infinite number of applications of rules Propagate and Decompose, and thus fail to terminate:

$$
\{x \not\doteq f(x)\} \rightsquigarrow \{f(x) \not\doteq f(f(x))\} \rightsquigarrow \{x \not\doteq f(x)\} \rightsquigarrow \ldots
$$

The side condition avoids this and the rule Occurs-check makes the unification fail: the solution would intuitively be the "infinite term"

$$
f(f(f(\ldots)))
$$

but those are not acceptable here.

In the worse case, the algorithm is exponential in time and space: hint, consider

$$
\{x_1 \not\doteq f(x_0, x_0), x_2 \not\doteq f(x_1, x_1), \ldots, x_n \not\doteq f(x_{n-1}, x_{n-1})\}
$$

but it performs well in practice.

5.4.4 Implementation. Terms can be implemented by the type

```
type term =
  | Var of string
  | App of string * term list
```

and we can check whether a variable x occurs in a term t, i.e. if $x \in \mathrm{FV}(t)$, with

```
let rec occurs x = function
  | Var y -> x = y
  | App (f, tt) -> List.exists (occurs x) tt
```

A substitution can be described as a list of pairs consisting of a variable (here, a string) and a term. It can be applied to a term thanks to the following function:

```
let rec app s = function
  | Var x -> (try List.assoc x s with Not_found -> Var x)
  | App (f, tt) -> App (f, List.map (app s) tt)
```

Unification can finally be performed by the following function, which takes as arguments the substitution being constructed (which is initially empty) and the equation system (a list of pairs of terms) and returns the most general unifier:

```
let rec unify s = function
  | (App (f, tt), App (g, uu))::e ->
    (* clash *)
    if f <> g then raise Not_unifiable
    (* decompose *)
    else unify s ((List.map2 (fun t u -> t, u) tt uu)@e)
  | (App (f, tt), Var x)::e ->
    (* orient *)
    unify s ((Var x, App (f, tt))::e)
  | (Var x, Var y)::e when x = y ->
    (* delete *)
    unify s e
  | (Var x, t)::e ->
    (* occurs check *)
    if occurs x t then raise Not_unifiable;
    (* propagate *)
    let t = app s t in
    let e = List.map (fun (t,u) -> app s t, app s u) e in
    let s = List.map (fun (x,t) -> x, app s t) s in
    unify ((x, t)::s) e
  | [] -> s
let unify = unify []
```

This function raises the exception `Not_unifiable` when the system has no solution. The unifier of example 5.4.3.2 can then be computed with

```
let s =
  let t =
    App ("f", [
      App ("a", []);
```

```
      App ("g", [Var "x"]);
      App ("g", [Var "x"])
    ]) in
  let u =
    App ("f", [
        App ("a", []);
        Var "y";
        App ("g", [App ("h", [Var "z"])])
      ]) in
  unify [t, u]
```

5.4.5 Efficient implementation. The major source of inefficiency in the previous algorithm is due to substitutions. In order to apply a substitution to a term, we have to go through the whole term to find variables to substitute, and moreover we have to apply a substitution to all the terms in the Propagate phase. Instead, if we are willing to use mutable structures, we can use the trick we have already encountered in section 4.4.3: we can have variables be references, so that we can modify their contents, and have them point to terms when we want to substitute those. This suggests that we should implement terms as

```
type term =
  | Var of var ref
  | App of string * term list
and var =
  | AVar of string
  | Link of term
```

A variable x is represented as `Var r` where `r` is a reference containing `AVar "x"`. If, later on, we want to substitute it with a term t, we can then modify the contents of `r` to `Link t`, which means that the variable has been replaced by t. When we do so, the contents of all the occurrences of the variable will thus be replaced at once.

While we could implement things in this way (similarly to section 4.4.3), we would like to explain another point and give a variant of this implementation. When encoding variables in this way, it is important that all the occurrences of the variable x contain the same reference, which is error prone: we have to ensure that, for a given variable name, the pointed memory cell is always the same. In most applications, the precise name of variables does not matter, since we are usually considering terms up to α-conversion. We can thus consider that the reference itself is the name of the variable, i.e. the name is the location in memory, which avoids the previous possibility for errors. Since two variables are now the same when their references are physically the same (i.e. when they point to the same memory cell, as opposed to having the same contents) and we should thus compare them using physical equality `==` instead of the usual extensional equality `=`. We can thus rather encode terms as

```
type term =
  | Var of term option ref
  | App of string * term list
```

A variable will initially be `Var r` with the reference `r` containing `None` (we do not use a string to indicate the name of the variable since the name is not relevant,

only the position in memory where the None is stored is), and substitution with a term t will amount to replacing this value by Some t. We can thus generate a fresh variable with

```
let var () = Var (ref None)
```

and the right notion of equality between terms is given by the following function

```
let rec eq t u =
  match t, u with
  | Var {contents = Some t}, u -> eq t u
  | t, Var {contents = Some u} -> eq t u
  | Var x, Var y -> x == y
  | App (f, tt), App (g, uu) ->
    f = g && List.for_all2 eq tt uu
  | _ -> false
```

where we use the fact that a reference is implemented in OCaml as a record with contents as only field, which is mutable. We can check whether a variable occurs in a term with

```
let rec occurs x = function
  | Var y -> x == y
  | App (f, tt) -> List.exists (occurs x) tt
```

using, as indicated above, physical equality to compare variables, and unification can be performed with

```
let rec unify t u =
  match t, u with
  | App (f, tt), App (g, uu) ->
    (* clash *)
    if f <> g then raise Not_unifiable
    (* decompose *)
    else List.iter2 unify tt uu
  (* follow links *)
  | Var {contents = Some t}, u -> unify t u
  | t, Var {contents = Some u} -> unify t u
  (* delete *)
  | Var x, Var y when x == y -> ()
  | Var x, u ->
    (* occurs check *)
    if occurs x u then raise Not_unifiable
    (* propagate *)
    else x := Some u
  | _, Var _ ->
    (* orient *)
    unify u t
```

The unifier of example 5.4.3.2 can then be computed with

```
let () =
  let x = var () in
```

```
let y = var () in
let z = var () in
let t =
  App ("f", [
      App ("a", []);
      App ("g", [x]);
      App ("g", [x])
    ]) in
let u =
  App ("f", [
      App ("a", []);
      y;
      App ("g", [App ("h", [z])])
    ]) in
unify t u
```

5.4.6 Resolution. A typical use of unification is to generalize the resolution technique of section 2.5.8 to first-order classical logic [Rob65].

Clausal form. In order to do so, we must first generalize the notion of clausal form:

  – a *literal* L is a predicate applied to terms or its negation

$$L ::= P(t_1, \ldots, t_n) \mid \neg P(t_1, \ldots, t_n)$$

  where P is a predicate of arity n and the t_i are terms,

  – a *clause* C is a disjunction of literals, i.e. a formula of the form

$$C ::= L_1 \vee L_2 \vee \ldots \vee L_k$$

We recall that a *theory* Θ on a given signature Σ is a set of closed formulas. Any theory can be put in clausal form in the following sense:

Proposition 5.4.6.1. Given a finite theory Θ on a signature Σ, there is a theory Θ' on a signature Σ' such that all the formulas in Θ' are clauses and the two theories Θ and Θ' are equisatisfiable.

Proof. The process of constructing Θ' from Θ is done in six steps.

  1. By lemma 5.1.7.4, we can replace any formula in Θ by an equivalent one in prenex form.

  2. By iterated use of proposition 5.2.3.10, we can replace any formula of Θ by an equisatisfiable one without existential quantification on a larger signature Σ'.

  3. By proposition 2.5.5.1, we can replace any formula in Θ, which is necessarily of the form $\forall x_1. \ldots \forall x_n.A$ where A is an arbitrary formula which does not contain any first-order quantification, by an equivalent one where A is a conjunction of disjunctions of literals, i.e. of the form

$$\forall x_1. \ldots \forall x_k. \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n_i} L_{i,j}$$

4. By repeated use of the equivalence

$$\forall x.(A \wedge B) \Leftrightarrow (\forall x.A) \wedge (\forall x.B)$$

we can replace every formula of Θ by a conjunction of universally quantified clauses

$$\bigwedge_{i=1}^{m} \forall x_1. \ldots. \forall x_k. \bigvee_{j=1}^{n_i} L_{i,j}$$

5. We can then replace every conjunction of clauses by all its universally quantified clauses

$$\forall x_1. \ldots. \forall x_k. \bigvee_{j=1}^{n_i} L_{i,j}$$

6. Finally, we can remove the universal quantifications in formulas if we suppose that all the universally quantified variables are distinct, see lemma 5.4.6.2 below.

The theory Θ' obtained in this way is equisatisfiable with Θ. $\qquad \square$

Lemma 5.4.6.2. Given formula $\forall x.A$ and a theory Θ, the theories $\Theta \cup \{\forall x.A\}$ and $\Theta \cup \{A\}$ are equisatisfiable, provided that $x \notin \mathrm{FV}(\Theta)$.

The resolution rule. We can assimilate a theory Γ in clausal form with a first order context. The *resolution rule* of section 2.5.8 can then be modified as follows in order to account for first-order:

$$\frac{\Gamma \vdash C \vee P(t_1, \ldots, t_n) \qquad \Gamma \vdash \neg P(u_1, \ldots, u_n) \vee D}{\Gamma \vdash (C \vee D)[\sigma]} \ (\mathrm{res})$$

where σ is the most general unifier of the equation system

$$\{t_1 \stackrel{?}{=} u_1, \ldots, t_n \stackrel{?}{=} u_n\}$$

Generalizing lemma 2.5.8.1, this rule is correct:

Lemma 5.4.6.3 (Correctness). If C can be deduced from Γ using the axiom and resolution rules then the sequent $\Gamma \vdash C$ is derivable in classical first-order logic.

Example 5.4.6.4. The standard example is the following one. We know that

– all men are mortal, and

– Socrates is a man,

which can be formalized as the theory

$$\{\forall x. \, \mathrm{man}(x) \Rightarrow \mathrm{mortal}(x), \mathrm{man}(\mathrm{Socrates})\}$$

in the signature with a constant symbol Socrates and with two unary predicates man and mortal. By proposition 5.4.6.1, we can put it in clausal form:

$$\{\neg \, \mathrm{man}(x) \vee \mathrm{mortal}(x), \mathrm{man}(\mathrm{Socrates})\}$$

We want to show that this entails that Socrates is mortal. As explained in lemma 2.5.8.8, this can be done using resolution by showing that adding

$$\neg \, \mathrm{mortal}(\mathrm{Socrates})$$

to the theory makes it inconsistent. And indeed, writing Γ for the resulting theory, we have

$$\cfrac{\cfrac{\overline{\Gamma \vdash \neg \, \mathrm{man}(x) \vee \mathrm{m}(x)} \; (\mathrm{ax}) \qquad \overline{\Gamma \vdash \mathrm{man}(\mathrm{S})} \; (\mathrm{ax})}{\Gamma \vdash \mathrm{m}(\mathrm{S})} \; (\mathrm{res}) \qquad \overline{\Gamma \vdash \neg \, \mathrm{m}(\mathrm{S})} \; (\mathrm{ax})}{\Gamma \vdash \bot} \; (\mathrm{res})$$

(we shortened mortal as m and Socrates as S).

The factoring rule. As is, the resolution rule is not complete (see example 5.4.6.6 below). We can however make the system complete by adding the following *factoring rule*

$$\cfrac{\Gamma \vdash C \vee P(t_1, \ldots, t_n) \vee P(u_1, \ldots, u_n)}{\Gamma \vdash (C \vee P(t_1, \ldots, t_n))[\sigma]} \; (\mathrm{fac})$$

where σ is the most general unifier of $\{t_1 \overset{?}{=} u_1, \ldots, t_n \overset{?}{=} u_n\}$. With this rule, the completeness theorem 2.5.8.7 generalizes as follows:

Theorem 5.4.6.5 (Refutation completeness). A set Γ of clauses is not satisfiable if and only if we can show $\Gamma \vdash \bot$ using axiom, resolution and factoring rules rules only.

Example 5.4.6.6. Given a unary predicate P, consider the theory

$$\Gamma = \{P(x) \vee P(y), \neg P(x) \vee \neg P(y)\}$$

which is not satisfiable. The resolution rule only allows us to deduce the clauses $P(x) \vee \neg P(x)$ and $P(y) \vee \neg P(y)$, from which we cannot deduce any other clause: without factoring, the resolution rule is not complete. With factoring, we can show that Γ is inconsistent by

$$\cfrac{\cfrac{\overline{\Gamma \vdash P(x) \vee P(y)} \; (\mathrm{ax})}{\Gamma \vdash P(x)} \; (\mathrm{fac}) \qquad \cfrac{\cfrac{\overline{\Gamma \vdash P(x) \vee P(y)} \; (\mathrm{ax})}{\Gamma \vdash P(x)} \; (\mathrm{fac}) \qquad \overline{\Gamma \vdash \neg P(x) \vee \neg P(y)} \; (\mathrm{ax})}{\Gamma \vdash \neg P(y)}}{\Gamma \vdash \bot} \; (\mathrm{res})$$

Instead of adding the factoring rule, in order to gain refutation completeness the resolution rule can also be modified in order to unify multiple literals at once.

Agda

6.1 What is Agda?

Agda is both a programming language and a proof assistant, originally developed by Norell in 2007. On the surface, it resembles a standard functional programming language such as OCaml or Haskell. However, it was designed with the Curry-Howard correspondence in mind, see chapter 4, extended to a much richer logic than propositional or first-order logic: it uses dependent types, which will be the object of chapter 8. This means that the types can express pretty much any proposition as a type and a program can be considered as a way of proving such a proposition. In this sense the language can also be considered as a *proof assistant*. We start by writing a type, which can be read as a formula, and gradually construct a program of this type, which can be read as a proof of the formula. The type checking algorithm of Agda will verify that the program actually admits the given type, i.e. that our proof is correct!

A first introduction to Agda is given in sections 6.2 and 6.3, inductive types are presented in section 6.4 for data types and section 6.5 for logical connectives, we discuss the formalization of equality in section 6.6, the use of Agda to prove the correctness of programs in section 6.7 and the issues related to termination in section 6.8.

6.1.1 Features of proof assistants. We shall first present some of the general features that Agda has or does not have. There is no room here for a detailed comparison with other proof assistants, the interested reader can find details in [Wie06] for instance. In passing, we will simply mention some difference with the main competitors, which are currently Coq and Lean, and operate similarly from our point of view. Other well-known proof assistants include ACL2, HOL Light, Isabelle, Mizar, PVS, etc.

No type inference. A first difference with functional programming languages (e.g. OCaml) is that the typing is so rich in proof assistants that there are no principal types and typability is undecidable. There is thus very limited support for type inference and we have to explicitly provide a type for all functions. The more precise the type for a function is, the longer implementing the program will take, but the stronger the guarantees will be. For instance, a sorting algorithm can be given the type

```
List A → List A
```

as usual, but also the type

```
List A → SortedList A
```

i.e. the type expresses the fact that the output is a sorted list (the type of sorted lists can be defined in the language). The second type is much more precise than

the first one, and it will be more involved to define a function of the second than of first type (although not considerably so).

Programs vs tactics. The Agda code looks pretty much like a program in a functional programming language. For instance, the proof of $A \times B \to A$ is, as expected a program which takes a pair (a, b) and returns a:

```
open import Data.Product

postulate A B : Set

proj : A × B → A
proj (a , b) = a
```

which is easily compared with the corresponding definition in the OCaml toplevel

```
# let proj (a , b) = a;;
val proj : 'a * 'b -> 'a = <fun>
```

On the contrary, Coq uses *tactics* which describe how to progress into the proof. The same proof in Coq would look like this:

```
Variables A B : Prop.

Theorem proj : (A * B) -> A.
Proof.
intro p.
elim  p.
intro a.
intro b.
exact a.
Qed.
```

It is not clear at all that it is implementing a projection, but the correspondence with the proof in natural deduction is obvious. The tactics precisely correspond to the rules, when read from bottom-up: the `intro` commands correspond to introduction of $\Rightarrow$ rules, `elim` to a variant of the usual elimination rule for $\wedge$, and `exact` to the axiom rule:

$$\cfrac{\cfrac{\cfrac{\cfrac{\overline{p : A \wedge B, a : A, b : B \vdash A} \; (\text{ax})}{p : A \wedge B, a : A \vdash B \Rightarrow A} \; (\Rightarrow_{\text{I}})}{p : A \wedge B \vdash A \Rightarrow B \Rightarrow A} \; (\Rightarrow_{\text{I}})}{p : A \wedge B \vdash A} \; (\wedge_{\text{E}})}{\vdash A \wedge B \Rightarrow A} \; (\Rightarrow_{\text{I}})$$

The difference between the two is mostly a matter of taste, both are quite convenient to use and have the same expressive power. The reason we chose to use Agda in this course is that it makes more clear the Curry-Howard correspondence, which is one of the main objects of this course.

Automation. There is one main advantage of using tactics over programs however: it allows more easily for automation, i.e. Coq can automatically build parts of the proofs for us. For instance, the previous example can be proved in essentially one line, which will automatically generate all the above steps:

```
Variables A B : Prop.

Theorem proj : (A * B) -> A.
Proof.
tauto.
Qed.
```

As a more convincing example, the following formula over integers

$$\forall m \in \mathbb{Z}.\forall n \in \mathbb{Z}.(1 + 2 \times m) \neq (n + n)$$

can also be proved in essentially one line:

```
Require Import Coq.ZArith.ZArith.
Require Import Coq.micromega.Lia.
Global Open Scope Z_scope.

Theorem thm : forall m n:Z, 1 + 2 * m <> n + n.
Proof.
intros; lia.
Qed.
```

(the `lia` tactic tries to automatically solve goals in linear integer arithmetic). If we had to do it by hand, we would have needed many steps, using small intermediate lemmas expressing facts such as $n + n = 2 \times n$, etc. Agda has only very limited support for automation, although it has been progressing recently using reflection.

Program extraction. A major feature of Coq is that the typing system allows to perform what is called *program extraction*: once the program is proved correct, one can extract the program (in OCaml) and forget about the parts which are present only to prove the correctness of the program. In contrast, the support for program extraction in Agda is less efficient and more experimental.

Correctness. It might seem obvious, but let us state this anyway: a proof assistant should be correct, in the sense that when it accepts a proof then the proof should actually be correct. Otherwise, it would be very easy to write a proof assistant:

```
let () =
  while true do
    let _ = read_line () in
    print_endline "Your proof is correct."
  done
```

We will see that sometimes the logic implemented in proof assistants is not consistent for very subtle reasons (for instance, in section 8.2.2): in this case, the program allows proving $\bot$ and thus any formula, and thus essentially amounts to the above although it is not obvious at all. For modern and well-developed proof assistants, we however have good reasons to trust that this is not the case, see below.

Small kernel. An important design point for a proof assistant is that it should have a small kernel, whose correctness ensures the correctness of the whole program: this is called the *de Bruijn criterion*. A proof assistant is made of a large number of lines of code (roughly 100 000 lines of Haskell for Agda and 225 000 lines of OCaml for Coq), those lines are written by humans and there is always the possibility that there is a bug in the proof assistant. For this reason, it is desirable that the part of the software that we really have to trust, its "kernel", which mainly consists in the typechecker, is as small as possible and isolated from the rest of the software, so that all the efforts to ensure correctness can be focused on this part. For instance, in Coq, a tactic can produce any proof in order to automate part of the reasoning: this is not really a problem because, in the end, the typechecker will ensure that the proof is correct, so that we do not have to trust the tactic. In Coq, the kernel is roughly 10% of the software; in Agda, the kernel is a bit larger, because it contains more features (dependent pattern matching in particular), which means that programming is easier in some aspects, but the trust that we have in the proof checker is a bit lower.

In order to have a small kernel, it is desirable to reuse as much as possible existing features; this principle is followed by most proof assistants. For instance in OCaml, there is a type `bool` of booleans, but those could already have been implemented using inductive types by

```
type bool = False | True
```

This is reasonable in OCaml to have a dedicated type for performance reasons but, in a proof assistant, this would mean more code to trust which is a bad thing: if we can encode a feature in some already existing feature, this is good. In Agda, booleans are actually implemented as above:

```
data Bool : Set where false true : Bool
```

as well as in Coq:

```
Inductive bool : Set := false : bool | true : bool.
```

Bootstrapping. A nice idea in order to gain confidence in the proof checker would be to *bootstrap* and prove its correctness inside itself: OCaml is programmed in OCaml, why couldn't we prove Agda in Agda? Gödel's second incompleteness theorem unfortunately shows that this is impossible. However, a fair amount can be done, and has been done in the case of Coq [BW97]: the part which is out of reach is to show the termination of Coq programs inside Coq (we already faced a similar situation, in the simpler case of Peano arithmetic, see section 5.2.5).

Termination. A proof assistant should be able to decide, in finite amount of time, whether a proof is correct or not. In order to do so, it has to be able to check that a given function will produce a value. For this reason, all the functions that you can write in proof assistants such as Agda are total: they always produce a result in a finite amount of time. In order to ensure this, heavy restrictions are imposed on the programs which can be implemented in proof assistants. Firstly, since all functions are total, the language is not Turing-complete: there are some programs that you can write in usual programming languages that you will not be able to write in a proof assistant. Fortunately, those are rare and typically arise when trying to bootstrap as explained above. Secondly, since the problem of deciding whether a function terminates or not is undecidable, the proof assistant actually implements conditions which ensure that accepted programs will terminate, but some terminating programs will actually get rejected for "no good reason". These issues are detailed in section 6.8.

6.1.2 Installation. In order to use Agda you will need two pieces of software: Agda itself and an editor which supports interacting with Agda. The recommended editor is Emacs.

Under Linux. On Ubuntu or Debian, installing Agda and Emacs is achieved by typing

```
sudo apt-get install agda emacs
```

(installation under most other distributions should be similar, by using the adequate package manager). Alternatively, in order to obtain a cutting-edge version, you can install `cabal` and type

```
cabal update
cabal install Agda
agda-mode setup
```

to compile the latest version of Agda.

VSCode. For people thinking that Emacs looks too old, a more modern-looking editor compatible with Agda is *Visual Studio Code*[1], which is available for most platforms. In order to activate Agda support, you should also install the dedicated Agda mode[2].

Under macOS and Windows. The preferred installation procedure under macOS and Windows changes from time to time. The latest one can be found in the documentation[3].

6.2 Getting started with Agda

6.2.1 Getting help. The first place to get started with Agda is the online documentation, which is quite well written:

---

[1]`https://code.visualstudio.com/`
[2]`https://marketplace.visualstudio.com/items?itemName=banacorn.agda-mode`
[3]`https://agda.readthedocs.io/en/latest/getting-started/installation.html`

```
https://agda.readthedocs.io/en/latest/
```

As usual you can also search on the web. In particular, there are also various forums such as Stackoverflow:

```
https://stackoverflow.com/questions/tagged/agda
```

A very good introduction to Agda is [WK19].

6.2.2 Shortcuts. When writing a proof in Agda, we do not have to write the whole program directly: this would be almost impossible in practice. The editor allows us to leave "holes" in proofs (written ?) and provides us with shortcuts which can be used in order to fill those holes and refine programs. Below we provide the shortcuts for the most helpful ones, writing C-x for the control key + the x key. They might seem a bit difficult to learn at first, but you will see that they are easy to get along, and we can live our whole Agda life with only six shortcuts.

Emacs. We should first recall the main Emacs shortcuts:

C-c C-s	save file
C-w	cut
M-w	copy
C-y	paste

Atom uses more standard ones.

Agda. The main shortcuts for Agda that we will need are the following ones, their use is explained below in section 6.2.6.

C-c C-l	typecheck and highlight the current file
C-c C-,	get information about the hole under the cursor
C-c C-space	give a solution
C-c C-c	case analysis on a variable
C-c C-r	refine the hole
C-c C-a	automatic fill
middle click	definition of the term

A complete list can be found in the online documentation. Shortcuts which are also sometimes useful are C-c C-. which is like C-c C-, but also shows the inferred type for the proposed term for a hole, and C-c C-n which normalizes a term (useful to test computations).

Symbols. Agda allows for using fancy UTF-8 symbols: those are entered using \ (backslash) followed by the name of the symbol (many names are shared with LaTeX). Most of them can be found in the documentation. The most useful ones are for logic

∧	\and	⊤	\top	→	\to	∀	\all	Π	\Pi	λ	\Gl
∨	\or	⊥	\bot	¬	\neg	∃	\ex	Σ	\Sigma	≡	\equiv

and some other useful ones are

ℕ	\bN	×	\times	≤	\le	∈	\in
⊎	\uplus	∷	\::	∎	\qed		

Indices and exponents such as in x_1 and x^1 are respectively typed `\_1` and `\^1`, and similarly for other.

6.2.3 The standard library. The standard library defines most of the expected data types. The default path is `/usr/share/agda-stdlib` and you are encouraged to have a look in there or in the online documentation. We list below some of the most useful modules.

Data types. The modules for common data types are:

`Data.Empty`	Empty type (⊥)
`Data.Unit`	Unit type (⊤)
`Data.Bool`	Booleans
`Data.Nat`	Natural numbers (ℕ)
`Data.List`	Lists
`Data.Vec`	Vectors (lists of given length)
`Data.Fin`	Types with finite number of elements

Other useful ones are : `Data.Integer` (integers), `Data.Float` (floating point numbers), `Data.Bin` (binary natural numbers), `Data.Rational` (rational numbers), `Data.String` (strings), `Data.Maybe` (option types), `Data.AVL` (balanced binary search trees).

Logic. Not much is defined in the core of the Agda language and most of the type constructors are also defined in the standard library:

`Data.Sum`	Sum types (⊎, ∨)
`Data.Product`	Product types (×, ∧, ∃, Σ)
`Relation.Nullary`	Negation (¬)
`Relation.Binary.PropositionalEquality`	Equality (≡)

Algebra. The standard library contains modules for useful algebraic structures in `Algebra.*`: monoids, rings, groups, lattices, etc.

6.2.4 Hello world. A mandatory example is the "hello world" program, see section 1.1.1. We can of course write it in Agda:

```
{-# OPTIONS --guardedness #-}

open import Level
open import IO

main : IO {a = 0ℓ} _
main = putStrLn "Hello, world!"
```

We however only give it for fun here: you will very rarely write such a program. A more realistic example is detailed in next section.

6.2.5 Our first proof. As a first proof, let's show that the propositional formula

$$A \wedge B \Rightarrow B \wedge A$$

is valid. By the Curry-Howard correspondence, we want a program of the type

$$A \times B \to B \times A$$

showing that $\times$ is commutative. In OCaml, we would have typed

```
# let prod_comm (a , b) = (b , a);;
val prod_comm : 'a * 'b -> 'b * 'a = <fun>
```

The full proof in Agda goes as follows:

```
open import Data.Product

-- The product is commutative
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B (a , b) = (b , a)
```

We should try to explain the various things we see there.

Importing modules. The programs in Agda, including the standard library, are organized in *modules* which are collections of functions dedicated to some feature. Here, we want to use the product, and therefore we have to import the corresponding module, which is called `Data.Product` in the standard library. In order to do so, we use the `open import` command which loads all its functions.

Comments. In Agda, comments start by `--` (two minus dashes), as in the second line.

Declaring functions. We are defining a function named `×-comm`. A function declaration always contains (at least) two lines. The first one is of the form

```
name : type
```

declaring that the function `name` will have the type `type`, and the second one is of the form

```
name a₁ ... aₙ = value
```

declaring that the function `name` takes arguments a_i and returns a given value.

Types. Let us detail the type

```
(A B : Set) → (A × B) → (B × A)
```

we have given to the function. As indicated above, in OCaml the type would have been

```
'a * 'b -> 'b * 'a
```

which means that for any types `'a` and `'b` the function can have the above type. In Agda, there is not such implicit universal quantification over types, which means that we have to do that by ourselves. We can do this because

1. we have the special type `Set` which is "the type of types" (we have *universes*),

2. we have the ability to name the arguments in types and use them in further types (we have *dependent types*).

The type of the function will thus read as: given arguments `A` and `B` of type `Set` (i.e. given types `A` and `B`), given a third argument of type `A × B`, we return a result of type `B × A`. The fact that the arguments `A` and `B` are grouped here is purely a syntactic convenience, and the above type is exactly the same as

```
(A : Set) → (B : Set) → (A × B) → (B × A)
```

Function definitions. The definition of the function is then the expected one

```
×-comm A B (a , b) = (b , a)
```

We take three arguments: `A`, `B` and a pair `(a , b)` and return the pair `(b , a)`. Note that the fact that we can write `(a , b)` for the third argument is because Agda allows definitions by *pattern matching* (just as OCaml): here, the product has only one constructor, the pair.

Spaces. A minor point, which is sometimes annoying at first, is that spaces for constructors are important: you have to write `(a , b)` and not `(a, b)` or `(a,b)`. This is because the syntax of Agda is really extensible (the notation for pairings is not built in for instance, it is defined in `Data.Product`!), which comes with some induced limitations. A side effect of this convention is that `a,b` is a perfectly legit variable name (but it is not necessarily a good idea to make heavy use of this opportunity).

Typesetting UTF-8 symbols. Since we want our proofs to look fancy, we have used some nice UTF-8 symbols: for instance "→" and "×". In the editor, such symbols are typed by commands such as `\to` or `\times` as indicated above, in section 6.2.2. There are usually text replacements (e.g. we could have written `->` and `*`), but those are not used much in Agda.

6.2.6 Our first proof, step by step. The above proof is very short, so that we could have typed it at once and then made sure that it typechecks, but even for moderately sized proofs, it is out of the question to write them in one go. Fortunately, we can input those gradually, by leaving "holes" in the proofs which are refined later. Let us detail how one would have done this proof step by step, in order to introduce all the shortcuts.

We begin by giving the type of the function and its declaration as

```
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B p = ?
```

We know that our function takes three arguments (`A`, `B` and `p`), which is obvious from the type, but we did not think hard enough yet of the result so that we have written `?` instead, which can be thought of as a "hole" in the proof. We can then typecheck the proof by typing `C-c C-l`. Basically, this makes sure that Agda is aware of what is in the editor (and report errors) so that you should use it whenever you have changed something in the file (outside a hole). Once we do that, the file is highlighted and changed to

```
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B p = { }0
```

The hole has been replaced by `{ }0`, meaning that Agda is waiting for some term here (the `0` is the number of the hole). Now, place your cursor in the hole. We can see the variables at our disposal (i.e. the context) by typing `C-c C-,`:

```
Goal: B × A
------------------------------------------------------------
p : A × B
B : Set
A : Set
```

This is useful to know where we are exactly in the proof: here we want to prove `B × A` with `A`, `B` and `p` of given types. Now, we want to reason by *case analysis* on `p`. We therefore use the shortcut `C-c C-c`, Agda then asks for the variable on which we want to reason by case on, in this case we reply `p` (and enter). The file is then changed to

```
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B (fst , snd) = { }0
```

Since the type of `p` is a product, `p` must be a pair and therefore Agda changes `p` to the pattern (`fst , snd`). Since we do not like the default names given by Agda to the variables, we rename `fst` to `a` and `snd` to `b`:

```
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B (a , b) = { }0
```

We should then do `C-c C-l` so that Agda knows of this change (remember that we have to do it each time we modify something outside a hole). Now, we place our cursor into the hole. By the same reasoning, the hole has a product as a type, so that it must be a pair. We therefore use the command `C-c C-r` which "refines" the hole, i.e. introduces the constructor if there is only one possible for the given type. The file is then changed to

```
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B (a , b) = { }1 , { }2
```

The hole was changed in a pair of two holes. In the hole `{ }1`, we know that the value should be `b`. We can therefore write `b` inside it and type `C-c C-space` to indicate that we have given the value to fill the hole:

```
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B (a , b) = b , { }2
```

We could do the same for the second hole (by giving `a`), but we get bored: this hole is of type `A` so that the only possible value for it was `a` anyway. Agda is actually able to find that if we type `C-c C-a`, which is the command for letting the proof assistant try to automatically fill a hole:

```
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B (a , b) = b , a
```

6.2.7 Our first proof, again. We would like to point out that these steps actually (secretly) correspond to constructing a proof. For simplicity, we suppose that A and B are two fixed types, this can be done by typing

```
postulate A B : Set
```

and consider the proof

```
×-comm : (A × B) → (B × A)
×-comm (a , b) = b , a
```

which is a small variant of previous one. We now explain that constructing this proof corresponds to constructing a proof in sequent calculus. As a general rule:

- doing a case split on a variable (`C-c C-c`) corresponds to performing a left rule (or an elimination rule in natural deduction),

- refining a hole (`C-c C-r`) corresponds to performing a right rule (or a introduction rule in natural deduction),

- providing a variable term (`C-c C-space`) corresponds to performing an axiom rule.

In figure 6.1, we have shown how the steps of our proof in Agda translate into the construction of the proof from bottom up, in sequent calculus. Also note that there is a perfect correspondence with respect to the Curry-Howard correspondence if we allow ourselves to put patterns instead of variables in the context:

$$
\cfrac{
  \cfrac{
    \cfrac{\overline{a:A, b:B \vdash b:B}\ (\text{ax}) \qquad \overline{a:A, b:B \vdash a:A}\ (\text{ax})}
    {a:A, b:B \vdash (b,a) : B \wedge A}\ (\wedge_{\mathrm{R}})
  }
  {(a,b) : A \wedge B \vdash (b,a) : B \wedge A}\ (\wedge_{\mathrm{L}})
}
{\vdash \lambda(a,b).(b,a) : A \wedge B \Rightarrow B \wedge A}\ (\Rightarrow_{\mathrm{R}})
$$

This correspondence has some defects in general [Kri09], which is why we do not detail it further here.

6.3 Basic Agda

In this section we present the main constructions which are present in the core of Agda, with the notable exception of inductive types which are described in sections 6.4 and 6.5.

Agda	Shortcut	Proof	Rule
`×-comm = { }0`	`C-c C-r`	$$\dfrac{\dfrac{?}{\vphantom{A}}}{\vdash A \wedge B \Rightarrow B \wedge A}$$	$(\Rightarrow_{\mathrm{R}})$
`×-comm p = { }0`	`C-c C-c p`	$$\dfrac{\dfrac{\dfrac{?}{A \wedge B \vdash B \wedge A}}{\vdash A \wedge B \Rightarrow B \wedge A}}{}$$	$(\wedge_{\mathrm{L}})$
`×-comm (a , b) = { }0`	`C-c C-r`	$$\dfrac{\dfrac{\dfrac{\dfrac{?}{A, B \vdash B \wedge A}}{A \wedge B \vdash B \wedge A}}{\vdash A \wedge B \Rightarrow B \wedge A}}{}$$	$(\wedge_{\mathrm{R}})$
`×-comm (a , b) = { }1 , { }2`	`b C-c C-s`	$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{?}{A, B \vdash B} \quad \dfrac{?}{A, B \vdash A}}{A, B \vdash B \wedge A}}{A \wedge B \vdash B \wedge A}}{\vdash A \wedge B \Rightarrow B \wedge A}}{}$$	(ax)
`×-comm (a , b) = b , { }2`	`a C-c C-s`	$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, B \vdash B} \quad \dfrac{?}{A, B \vdash A}}{A, B \vdash B \wedge A}}{A \wedge B \vdash B \wedge A}}{\vdash A \wedge B \Rightarrow B \wedge A}}{}$$	(ax)
`×-comm (a , b) = b , a`		$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, B \vdash B} \quad \dfrac{}{A, B \vdash A}}{A, B \vdash B \wedge A}}{A \wedge B \vdash B \wedge A}}{\vdash A \wedge B \Rightarrow B \wedge A}$$	

Figure 6.1: Agda proofs and sequent proofs.

6.3.1 The type of types. In Agda, there is by default a type named `Set`, which can be thought of as the type of all types: an element of type `Set` is a type.

6.3.2 Arrow types. In Agda, we have the possibility of forming function types: given types `A` and `B`, one can form the type

$$A → B$$

of functions taking an argument of type `A` and returning a value of type `B`. For instance, the function `isEven` which determines whether a natural number is boolean will be given the type

```
isEven : ℕ → Bool
```

Type constructors. Functions in Agda can operate on types. For instance, the type of lists is a *type constructor*: it is a function which takes a type `A` as argument and produces a new type `List A`, the type of lists whose elements are of type `A`. We can thus give it the type

```
List : Set → Set
```

The type `List A` can also be seen as a type which is parametrized by another type, just as in OCaml the type `'a list` of lists is parametrized by the type `'a`.

Named arguments. In Agda, we can give a name to the arguments in types, e.g. we can give the name `x` to `A` and consider the type

$$(x : A) → B$$

For instance, the even function could also have been given the type

```
isEven : (x : ℕ) → Bool
```

However, the added power comes from the fact that the type `B` is also allowed to make use of the variable `x`. For instance, the function which constructs a singleton list of some type can be given the following type (see section 6.3.3 for the full definition of this function):

```
singleton : (A : Set) → A → List A
```

Both the second argument and the result use the type `A` which is given as first argument. Such a type is called a *dependent type*: it can depend on a value, which is given as an argument.

Universal quantification. Another way to read the type `(x : A) → B` is as a *universal quantification*: it corresponds to what we would have previously written

$$\forall x \in A.B$$

For instance, we can define the type of equalities between two elements of a given type `A` by

```
eq : (A : Set) → A → A → Set
```

and a proof that this equality is reflexive is given the type

```
refl : (A : Set) → (x : A) → eq A x x
```

which corresponds to the usual formula

$$\forall A.\forall x \in A.x = x$$

Implicit arguments. Sometimes, some arguments can be deduced from the type of other arguments. For instance, in the `singleton` function above, A is the type of the second argument. In this case, we can make the first argument *implicit*, which means that we will not have to write it and we will let Agda guess it instead. This is done by using curly brackets in the type

```
singleton : {A : Set} → A → List A
```

This allows us to simply write

```
singleton 3
```

and Agda will be able to find out that A has to be ℕ, since this is the type of 3. In case we want to specify the implicit argument, we have to use the same brackets:

```
singleton {ℕ} 3
```

Another way of having Agda make a guess is to use _ which a placeholder that has to be filled automatically by Agda. For instance, we could try to let Agda guess the type of A (which is Set) by declaring

```
singleton : {A : _} → A → List A
```

which can equivalently be written

```
singleton : ∀ {A} → A → List A
```

6.3.3 Functions. As indicated in section 6.2.5, a function definition begins with a line specifying the type of the function, followed by the definition of the function itself. For instance, the singleton function which takes an element x of some arbitrary type A and returns the list with x as the only element, can be defined as

```
singleton : (A : Set) → A → List A
singleton A x = x :: []
```

or as

```
singleton : {A : Set} → A → List A
singleton x = x :: []
```

or as

```
singleton : {A : Set} → A → List A
singleton = λ x → x :: []
```

In the second variant, A is an implicit argument. In the third variant, we illustrate the fact that we can use λ-abstractions to define anonymous functions.

Infix notations. In function names, underscores (`_`) are handled as places where the arguments should be put, which allows to easily define infix operators. For instance, we can define the addition with type

```
_+_ : ℕ → ℕ → ℕ
```

and then use it as

```
3 + 2
```

The prefix notation is still available though, in case it is needed:

```
_+_ 3 2
```

The priorities of binary operators can be specified by commands such as

```
infix 6 _+_
```

which states that the priority of addition is `6` (the higher the number, the higher the priority). Operations can also be specified to be left (resp. right) associative by replacing `infix` by `infixl` (resp. `infixr`). For instance, addition and multiplications are usually given priorities

```
infixl 6 _+_
infixl 7 _*_
```

so that the expression

$$2 + 3 + 5 * 2$$

is implicitly bracketed as

$$(2 + 3) + (5 * 2)$$

Auxiliary functions. In the definition of a function, it is possible to use auxiliary function definitions using the `where` keyword. For instance, we can define the function f which computes the fourth power of a natural number, i.e. $f(x) = x^4$, by using the square function as an auxiliary function, i.e. $f(x) = (x^2)^2$, as follows:

```
fourth : ℕ → ℕ
fourth n = square (square n)
  where
  square : ℕ → ℕ
  square n = n * n
```

Here, we define the `fourth` function in terms of the auxiliary function `square`, which is defined afterwards, preceded by the `where` keyword.

6.3.4 Postulates. It rarely happens that we need to assume the existence of a term of a given type without any hope of proving it: this is typically the case for axioms. This can be achieved by the `postulate` keyword. For instance, in order to work in classical logic, we can assume the law of excluded middle with

```
postulate lem : (A : Set) → ¬ A ⊎ A
```

These should be avoided as much as possible because postulates will not compute: if we apply `lem` to an actual type `A`, it will not reduce to either `¬ A` or `A`, as we would expect for a coproduct, see section 6.5.6: how could Agda possibly know which one is the right one?

6.3.5 Records. Records in Agda are pretty similar to those in other language (e.g. OCaml) and will not be used much here. In order to illustrate the syntax, we provide here an implementation of pairs using records:

```
record Pair (A B : Set) : Set where
  field
    fst : A
    snd : B

make-pair : {A B : Set} → A → B → Pair A B
make-pair a b = record { fst = a ; snd = b }

proj₁ : {A B : Set} → Pair A B → A
proj₁ p = Pair.fst p
```

6.3.6 Modules. A module is a collection of functions. It can be declared by putting

```
module Name where
```

at the beginning of the file, where `Name` is the name of the module and should match the name of the file. The functions of another module can be used by issuing the command

```
open import Name
```

which will expose all the functions of the module `Name`. After this command, the modifiers `hiding (...)` or `renaming (... to ...)` can be used in order to hide or rename some of the functions.

6.4 Inductive types: data

Inductive types are the main way of defining new types in Agda. Apart from a few exceptions (such as → and `Set` mentioned above), all the usual types are defined using this mechanism in the standard library, including usual data types and logical connectives; we first focus on data types in this section. An inductive type `T` is declared using a statement of the form

```
data T : A where
  cons₁ : A₁ → ... → Aᵢ → T
  ...
  consₙ : B₁ → ... → Bⱼ → T
```

which declares that `T` is an inductive type, whose type is `A`, with constructors $\text{cons}_1, \ldots, \text{cons}_n$. For each constructor, the line begins with two blank spaces, followed by the name of the constructor, and ends with the type of the constructor. Each constructor takes an arbitrary number of arguments and has `T` as return type. Since the type `T` we are defining is itself a type, `A` is usually `Set`, although some more general inductive types are supported (for instance, they can depend on some other types, see section 6.4.7).

6.4.1 Natural numbers. As a first example, the natural numbers are defined as the inductive type ℕ in the module `Data.Nat` by

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

The first constructor is `zero`, which does not take any argument, and the second constructor is `suc`, which takes a natural number as argument. A value of type ℕ is

```
   zero      suc zero      suc (suc zero)      suc (suc (suc zero))
```

and so on. As a convenience, the usual notation for natural numbers is also supported and we can write `2` as a shorthand for `suc (suc zero)`.

6.4.2 Pattern matching. The way one typically uses elements of an inductive type is by pattern matching: it allows inspecting a value of an inductive type and return a result depending on the constructor of the value. As explained above, the cases are usually generated by using the `C-c C-c` shortcut which instructs the editor to perform case analysis on some variable. For instance, in order to define the predecessor function, we start with

```
pred : ℕ → ℕ
pred n = ?
```

then, by `C-c C-c` we indicate that we want to reason by case analysis on `n`, which turns the code into

```
pred : ℕ → ℕ
pred zero    = ?
pred (suc n) = ?
```

We now have to give the result of the function when the argument is `zero` (by convention the predecessor of 0 is 0) and when the argument is `suc n`, where `n` is a natural number. We can finally fill in the holes in order to define the predecessor:

```
pred : ℕ → ℕ
pred zero    = zero
pred (suc n) = n
```

Of course, pattern matching also works with multiple arguments and we can define addition by

```
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
```

This definition can be tested by defining

```
t = 3 + 2
```

and use `C-c C-n` to normalize `t` (which give `5` as answer). Subtraction can be defined similarly by

```
_∸_ : ℕ → ℕ → ℕ
zero  ∸ n     = zero
suc m ∸ zero  = suc m
suc m ∸ suc n = m ∸ n
```

(by convention $m - n = 0$ when $m < n$) and multiplication by

```
_*_ : ℕ → ℕ → ℕ
zero  * n = zero
suc m * n = (m * n) + n
```

Matching with other values. It is sometimes useful to define a function by case analysis on a value which is not an argument. In this case, we can use the with keyword followed by the value we want to match on. This value can then be matched as an extra argument, which has to be separated from the other argument by a symbol |. For instance, the modulo function on natural numbers can be defined by induction on the second argument by the following definition:

$$m \bmod n = \begin{cases} m & \text{if } m < n \\ (m - n) \bmod n & \text{otherwise} \end{cases}$$

Here, we do not want to reason directly by induction on n, which would force us to distinguish the case where n is zero or a successor, but rather on whether $m < n$ holds or not. This can be achieved by matching on m <? n which will either be yes _ or no _ depending on whether $m < n$ or $m \not< n$ (the arguments of those constructors are not important for the moment and will be detailed in section 6.5.6).

We begin our definition as usual with

```
_mod_ : ℕ → ℕ → ℕ
m mod n = ?
```

Since we want to match on m <? n, we use the with keyword in order to match on it additionally to the arguments:

```
_mod_ : ℕ → ℕ → ℕ
m mod n with m <? n
m mod n | p = ?
```

and we can then reason by case analysis on p. Incidentally, we can avoid typing again the match on the arguments of the function by simply writing "...":

```
_mod_ : ℕ → ℕ → ℕ
m mod n with m <? n
... | p = ?
```

At this point, we reason by case analysis on p (with C-c C-c p) which will produce two cases depending on the value of p:

```
_mod_ : ℕ → ℕ → ℕ
m mod n with m <? n
m mod n | yes _ = ?
m mod n | no  _ = ?
```

We can finally fill those two cases, as indicated by the above formula:

```
_mod_ : ℕ → ℕ → ℕ
m mod n with m <? n
m mod n | yes _ = m
m mod n | no  _ = (m ∸ n) mod n
```

As a side note, if you actually try the above definition in Agda, you will see that it gets rejected because it is not clear for Agda that it is actually terminating. The actual definition is slightly more involved because of this, see section 6.8.

Empty pattern matching. Some inductive types do not have any element. For instance, we can define the empty type ⊥ as

```
data ⊥ : Set where
```

(this is an inductive type with no elements). When performing pattern matching on elements of this type there can be no match. In order to represent this, Agda uses the pattern (), which means that no such pattern can happen. For instance, one can show that if we have an element of type ⊥ then we have an element of an arbitrary type A as follows:

```
⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()
```

Of course, since the type A is arbitrary, there is no way for us in the proof to actually exhibit a term of this type. But we do not have to: the pattern () states that there are no cases to handle when matching on the argument of type ⊥, so that we are done.

It might seem at first that this is not so useful, unless one insists on using the type ⊥ (which is actually done quite often since negation is defined using it as you can expect). This is not so because there are many less obvious ways of constructing empty inductive types in Agda. For instance, the type zero ≡ suc zero of equalities between 0 and 1 is also an empty inductive type.

Anonymous pattern matching. Anonymous functions can be defined by pattern matching, although the syntax is slightly different from what one would expect: we need to put curly brackets before the arguments, and cases are separated by semicolons:

```
λ { x → ... ; ... }
```

For instance, the predecessor can be defined as an anonymous function by

```
pred : ℕ → ℕ
pred = λ { zero → zero ; (suc n) → n }
```

6.4.3 The induction principle. We would now like to briefly mention that pattern matching in Agda corresponds to the presence of a *recursion* principle (for non-dependent functions) or of an *induction* principle (for dependent functions).

For instance, if we define a function f from natural numbers to some type A, we will typically define it using pattern matching by

```
f : ℕ → A
f zero    = t
f (suc n) = u'
```

where `t` and `u'` are terms of type `A`. Here, `u'` might make use of the natural number `n` provided as the argument, as well as the result of the recursive call `f n`: we can suppose that `u'` is of the form `u n (f n)` for some function `u` of type `ℕ → A → A`. Any such terms `t` and `u` will give rise to a function of type `ℕ → A` in this way, and the *recursion principle* expresses this through a function which takes two arguments (of type `A` and `ℕ → A → A`, respectively corresponding to `t` and `u`) and produces the resulting function:

```
rec : {A : Set} → A → (ℕ → A → A) → ℕ → A
rec t u zero    = t
rec t u (suc n) = u n (rec t u n)
```

This is precisely the recursor we have already met when adding natural numbers to simply typed λ-calculus in section 4.3.6. Moreover, any function of type `ℕ → A` defined using pattern matching can be defined using this function instead: this recursion function encapsulates all the expressive power of pattern matching that can be used in order to define non-dependent functions on natural numbers. For instance, the predecessor function would be defined as

```
pred : ℕ → ℕ
pred = rec zero (λ n _ → n)
```

From a logical point of view, the recursion principle corresponds to the elimination rule: for this reason it is also sometimes also called an *eliminator*.

Pattern matching in Agda is more powerful than this however: it can also be used in order to define functions where the return type depends on the argument. This means that we now consider functions of the form

```
f : (n : ℕ) → P n
f zero    = t
f (suc n) = u n (f n)
```

where `P n` is a type which depends on `n`, or equivalently `P` is a predicate, of type `ℕ → Set`: here, `t` is of type `P zero` and `u n (f n)` is of type `P (suc n)`. The corresponding dependent variant of the recursion principle is called the *induction principle* and is the following one:

```
rec : (P : ℕ → Set) → P zero →
      ((n : ℕ) → P n → P (suc n)) → (n : ℕ) → P n
rec P Pz Ps zero    = Pz
rec P Pz Ps (suc n) = Ps n (rec P Pz Ps n)
```

Given

- a predicate `P`,

- an element `t` of `P zero`, and

- a function `u` of type `ℕ → P n → P (suc n)`,

this function allows us to construct a function of type `(n : ℕ) → P n`. If, following the Curry-Howard correspondence, we read the type as a logical formula (see section 6.5), we precisely recover the usual induction principle over natural numbers:

$$P(0) \Rightarrow (\forall n \in \mathbb{N}.P(n) \Rightarrow P(n+1)) \Rightarrow \forall n \in \mathbb{N}.P(n)$$

For instance, the following proof by induction that $n + 0 = n$ for every natural number n

```
+-zero : (n : ℕ) → n + zero ≡ n
+-zero zero    = refl
+-zero (suc n) = cong suc (+-zero n)
```

can be expressed as follows using the induction principle:

```
+-zero : (n : ℕ) → n + zero ≡ n
+-zero = rec (λ n → n + zero ≡ n) refl (λ n p → cong suc p)
```

6.4.4 Booleans. The type of *booleans* is defined in `Data.Bool` by

```
data Bool : Set where
  false : Bool
  true  : Bool
```

so that, for instance, boolean negation is defined by

```
neg : Bool → Bool
neg false = true
neg true  = false
```

and conjunction by

```
_∧_ : Bool → Bool → Bool
false ∧ _     = false
true  ∧ false = false
true  ∧ true  = true
```

In Agda, even conditional branchings are defined by pattern matching:

```
if_then_else_ : {A : Set} → Bool → A → A → A
if false then x else y = x
if true  then x else y = y
```

Finally, the induction principle for booleans is

```
Bool-rec : (P : Bool → Set) → P false → P true →
           (b : Bool) → P b
Bool-rec P Pf Pt false = Pf
Bool-rec P Pf Pt true  = Pt
```

6.4.5 Lists. Lists are defined in `Data.List` by

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A
```

Here, "::" is one UTF-8 symbol (entered with `\::`) and not two colons. As indicated above, the type `List` depends on another type `A`, called the *parameter* of the inductive type. The resulting type is thus called *parametric type* or a *type constructor*. The usual functions are defined as usual by induction, for instance, we can define the function which associates to a list its length by

```
length : {A : Set} → List A → ℕ
length []       = 0
length (x :: l) = suc (length l)
```

the function which maps a function to every element of a list by

```
map : {A B : Set} → (A → B) → List A → List B
map f []       = []
map f (x :: l) = f x :: map f l
```

or the function which concatenates two lists by

```
_++_ : {A : Set} → List A → List A → List A
[]       ++ l' = l'
(x :: l) ++ l' = x :: (l ++ l')
```

Finally, the induction principle for lists is:

```
List-rec : {A : Set} → (P : List A → Set) → P [] →
  ((x : A) → (xs : List A) → P xs → P (x :: xs)) →
  (xs : List A) → P xs
List-rec P Pe Pc [] = Pe
List-rec P Pe Pc (x :: xs) = Pc x xs (List-rec P Pe Pc xs)
```

6.4.6 Options. *Option types* are defined in `Data.Maybe` by

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A
```

A value of this type is thus either `nothing` or `just x` for some value x of type A. The type `Maybe A` can thus be seen as the type A extended with one new value, `nothing` (it corresponds to `option` types of OCaml, see section 1.3.4). It is often useful in order to accommodate for exceptional values (where we would use "NULL pointers" in other languages). For instance, the function returning the head of a list is not defined when the list is empty. It can be given the following definition:

```
head : {A : Set} → List A → Maybe A
head []       = nothing
head (x :: l) = just x
```

This function is a bit cumbersome to use: each time we have to test whether the result is `nothing` or not (monads [Mog91] might be of some help here though). A more elegant solution is provided below.

6.4.7 Vectors. A *vector* is a list of given length. The type of vectors is defined in `Data.Vec` by

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _∷_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

An element of `Vec A n` can be seen as a list whose elements are of type `A` and whose length is `n`. In this type, we thus have both a *parameter* `A` of type `Set` and an *index* of type `ℕ`, corresponding to the length of the vector, indicated by the fact that the return type is `ℕ → Set`. Indices are roughly the same as parameters, except that they can vary with constructors, as seen above: the constructor `[]` produces a vector of length `zero`, whereas the constructor `_∷_` a list of length `suc n`. It is "pure coincidence" if the names of the constructors are the same as for lists: they have nothing to do with those and could have been named differently (however, people chose to name them in the same way because vectors are usually used as a replacement for lists).

Dependent types. It should be observed that the type `Vec A n` of vectors depends on a term `n`, the natural number indicating its length: this is a defining feature of dependent types. We can also define functions such that the type of the result depends on the argument. For instance, we have the following function, building a vector containing n occurrences of a given value:

```
replicate : {A : Set} → A → (n : ℕ) → Vec A n
replicate x zero    = []
replicate x (suc n) = x ∷ replicate x n
```

Dependent pattern matching. Another natural function on this type is the function returning the head of the list:

```
head : {n : ℕ} {A : Set} → Vec A (suc n) → A
head (x ∷ xs) = x
```

This is a good illustration of the *dependent pattern matching* present in Agda. Since the argument is a list of type `Vec A (suc n)`, Agda automatically infers that this function will never be applied to an empty list, because it cannot have such a type, thus avoiding the problem we had when defining the same function on lists in section 6.4.6.

Convertibility. Even though the type is more informative than the one of lists, typical functions are not significantly harder to write. For instance, the concatenation of vectors is comparable to the one of lists:

```
_++_ : {m n : ℕ} {A : Set} → Vec A m → Vec A n → Vec A (m + n)
[]       ++ l  = l
(x ∷ l) ++ l' = x ∷ (l ++ l')
```

Looking closely at the first case of the pattern matching, we can note that the result `l` we are providing is of type `Vec A n` whereas the type of the function indicates that we should provide a result of type `Vec A (zero + n)`. This illustrates the fact that Agda is able to compare types up to β-reduction on terms (`zero + n` reduces to `n`): we can never distinguish between two β-convertible terms.

Induction principle. The induction principle for vectors is:

```
Vec-rec : {A : Set} → (P : {n : ℕ} → Vec A n → Set) → P [] →
  ({n : ℕ} (x : A) (xs : Vec A n) → P xs → P (x :: xs)) →
  {n : ℕ} → (xs : Vec A n) → P xs
Vec-rec P Pe Pc [] = Pe
Vec-rec P Pe Pc (x :: xs) = Pc x xs (Vec-rec P Pe Pc xs)
```

Indices instead of parameters. In the definition of vectors, we could have used an index instead of a parameter for the type A:

```
data Vec : Set → ℕ → Set where
  []  : {A : Set} → Vec A zero
  _::_ : {A : Set} {n : ℕ} (x : A) (xs : Vec A n) → Vec A (suc n)
```

This is a general fact: we can always encode a parameter as an index. However, it is recommended to use parameters whenever possible, because Agda handles them more efficiently.

Also, with the above definition, the induction principle is slightly different:

```
Vec-rec : (P : {A : Set} {n : ℕ} → Vec A n → Set) →
  ({A : Set} → P {A} []) →
  ({A : Set} {n : ℕ} (x : A) (xs : Vec A n) → P xs → P (x :: xs)) →
  {A : Set} → {n : ℕ} → (xs : Vec A n) → P xs
Vec-rec P Pe Pc [] = Pe
Vec-rec P Pe Pc (x :: xs) = Pc x xs (Vec-rec P Pe Pc xs)
```

6.4.8 Finite sets. In section 6.4.4, we have defined the set of booleans, which contains two elements, and clearly we could have defined a set with n elements for any fixed natural number n. For instance, the following type has four elements:

```
data Four : Set where
  a : Four
  b : Four
  c : Four
  d : Four
```

In fact, we can define, once for all, a type `Fin` n which depends on a natural number n and has n elements. The definition is done in `Data.Fin` by

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

Looking at it, we can see that `Fin` n is essentially the collection of natural numbers restricted to

$$\text{Fin } n = \{0, \ldots, n-1\}$$

Namely, the above inductive type corresponds to the following inductive set-theoretic definition:

$$\text{Fin } 0 = \emptyset$$

$$\text{Fin } (n+1) = \{0\} \cup \{i+1 \mid i \in \text{Fin } n\}$$

As for vectors, the fact that the constructors have the same name as for natural numbers is "pure coincidence": the elements of Fin n are not elements of ℕ, although there is obviously a canonical mapping:

```
toℕ : {n : ℕ} → Fin n → ℕ
toℕ zero    = zero
toℕ (suc i) = suc (toℕ i)
```

Some black magic in Agda allows it to determine, using types, whether we are using the constructors of Fin or those of ℕ.

The lookup function. The type Fin n is typically used to index elements over finite sets. For instance, consider the lookup function, which returns the i-th element of a vector l of length n. Clearly, this function is only well defined when $i < n$, i.e. when i belongs to Fin n. We can define this function as follows:

```
lookup : {n : ℕ} {A : Set} → Fin n → Vec A n → A
lookup zero    (x :: l) = x
lookup (suc i) (x :: l) = lookup i l
```

The typing ensures that the index will always be such that the function is well-defined, i.e. that we will never request an element outside the boundaries of the vector.

Let us present other possible implementations of this function, using natural numbers as the type of i, in order to show that they are more involved and less elegant. Since the function is not defined for every natural number i, a first possibility would be to have a return value of type Maybe A, where nothing would indicate that the function is not defined:

```
lookup : ℕ → {A : Set} {n : ℕ} → Vec A n → Maybe A
lookup zero    []      = nothing
lookup zero    (x :: l) = just x
lookup (suc i) []      = nothing
lookup (suc i) (x :: l) = lookup i l
```

This is quite heavy to use in practice, because we have to account for the possibility that the function is not defined each time we use it. Another option could be to add as argument a proof of i < n, ensuring that the index is not out of bounds. This is more acceptable in practice, but the definition is not as direct as the one above:

```
lookup : {i n : ℕ} {A : Set} → i < n → Vec A n → A
lookup {i}     {.0}       () []
lookup {zero}  {.(suc _)} i<n (x :: l) = x
lookup {suc i} {.(suc _)} i<n (x :: l) = lookup (≤-pred i<n) l
```

6.4.9 Integers. The type of integers can be defined essentially by taking two copies of the natural numbers: one corresponding to the positive integers and the other to the negative integers. If we proceed in this way, we however have two representations of zero (as 0 or −0), which should be identified. In order to avoid this problem, one of the two copies (here, the negative integers) is shifted by one. We thus define the type of integers as

```
data ℤ : Set where
  pos :    ℕ → ℤ
  negsuc : ℕ → ℤ
```

The encoding of 0 is `pos 0`, 3 is `pos 3` and -5 is `negsuc 1` (note the shift by one). The successor function `suc` is defined by induction by

```
suc : ℤ → ℤ
suc (pos n)            = pos (ℕ.suc n)
suc (negsuc ℕ.zero)    = pos 0
suc (negsuc (ℕ.suc n)) = negsuc n
```

and the predecessor `pred` is defined similarly. Finally, addition can be implemented using those by

```
_+_ : ℤ → ℤ → ℤ
pos ℕ.zero        + n = n
pos (ℕ.suc m)     + n = suc (pos m + n)
negsuc ℕ.zero     + n = pred n
negsuc (ℕ.suc m)  + n = pred (negsuc m + n)
```

6.5 Inductive types: logic

We have seen that inductive types can be used in order to implement usual data types (and more). We now explain that they can also be used to implement usual constructions on the logical side: though the Curry-Howard correspondence, types can be read as logical formulas (and a program of a given type as a proof of its type). We establish the translation between the two in this section. In this way, Agda provides a formal framework in which proofs can be formalized, as hinted in section 1.5, and we will see that is much richer than the simply-typed λ-calculus presented in chapter 4.

6.5.1 Implication. The first logical connective we have at our disposal is implication, which corresponds to the arrow → in types. For instance, the classical formulas

$$A \Rightarrow B \Rightarrow A \qquad\qquad (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$$

can respectively be proved by

```
K : {A B : Set} → A → B → A
K x y = x
```

and

```
S : {A B C : Set} → (A → B → C) → (A → B) → A → C
S g f x = g x (f x)
```

6.5.2 Product. *Products* are defined in `Data.Product` by

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

The first projection is defined as

```
proj₁ : {A B : Set} → A × B → A
proj₁ (a , b) = a
```

and the second projection is defined similarly. The projections are named `proj₁` and `proj₂` in the standard library, even though we sometimes like to rename them as `fst` and `snd`. From a logical point of view, a product corresponds to conjunction. For instance, a proof of the formula $A \wedge B \Rightarrow B \wedge A$, expressing the commutativity of conjunction, was given in section 6.2.5. As another example, currying (see section 4.3.1) can be shown by

```
×-→ : {A B C : Set} → (A × B → C) → (A → B → C)
×-→ f x y = f (x , y)
```

and

```
→-× : {A B C : Set} → (A → B → C) → (A × B → C)
→-× f (x , y) = f x y
```

Introduction rule. It can be observed that the constructor corresponds to the introduction rule for conjunction:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \ (\wedge_{\mathrm{I}})$$

This is a general fact: when defining logical connectives with inductive types, constructors correspond to introduction rules. We see below that the elimination rule corresponds to the associated induction principle.

Induction principle. The induction principle for products is

```
×-ind : {A B : Set} → (P : A × B → Set) →
        ((x : A) → (y : B) → P (x , y)) → (p : A × B) → P p
×-ind P Pp (x , y) = Pp x y
```

In the case where `P` does not depend on its argument, the above induction principle implies the following simpler principle

```
×-rec : {A B : Set} → (P : Set) → (A → B → P) → A × B → P
×-rec P Pp (x , y) = Pp x y
```

which corresponds to the elimination rule for conjunction:

$$\frac{\Gamma, A, B \vdash P \qquad \Gamma \vdash A \wedge B}{\Gamma \vdash P} \ (\wedge_{\mathrm{E}})$$

Namely, it states that if the premises are true then the conclusion is also true. The dependent induction principle corresponds to the elimination rule in dependent types, as we will see in section 8.3.3.

6.5.3 Unit type. The *unit type* is defined in `Data.Unit` by

```
data ⊤ : Set where
  tt : ⊤
```

From a logical point of view, the type corresponds to truth and the constructor to the introduction rule

$$\frac{}{\Gamma \vdash \top} \; (\top_{\mathrm{I}})$$

Its induction principle is

```
⊤-rec : (P : ⊤ → Set) → P tt → (t : ⊤) → P t
⊤-rec P Ptt tt = Ptt
```

We know from logic that there is no elimination rule associated to truth. We can however write the rule which corresponds to this induction principle:

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash \top}{\Gamma \vdash P} \; (\top_{\mathrm{E}})$$

This is not very interesting from a logical point of view: if we know that P holds and $\top$ holds then we can deduce that P holds, which we already knew.

6.5.4 Empty type. The *empty type* is defined in `Data.Empty` by

```
data ⊥ : Set where
```

and corresponds to falsity. It has no constructor, thus no introduction rule. The associated induction principle is

```
⊥-elim : (P : ⊥ → Set) → (x : ⊥) → P x
⊥-elim P ()
```

The non-dependent variant of this principle

```
⊥-elim : (P : Set) → ⊥ → P
⊥-elim P ()
```

corresponds to the explosion principle, which is the associated elimination rule

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P} \; (\bot_{\mathrm{E}})$$

We recall from section 6.4.2 that `()` is the empty pattern in Agda, which indicates here that there are no cases to handle when matching on a value of type `⊥`.

6.5.5 Negation. As expected, negation is defined in `Relation.Nullary` by

```
¬ : Set → Set
¬ A = A → ⊥
```

For instance, the formula $A \Rightarrow \neg\neg A$ can be proved with

```
nni : {A : Set} → A → ¬ (¬ A)
nni x f = f x
```

6.5.6 Coproduct. *Coproducts* (or *sums*) are defined in `Data.Sum` by

```
data _⊎_ (A : Set) (B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

The constructor `inj₁` (resp. `inj₂`) is called the *injection* of `A` (resp. `B`) into `A ⊎ B`. The notation comes from the fact that the coproduct `A ⊎ B` corresponds to the disjoint unions if we see the types `A` and `B` as sets. Logically, coproduct corresponds to disjunction. As an illustration, the commutativity of disjunction is shown by

```
⊎-comm : (A B : Set) → A ⊎ B → B ⊎ A
⊎-comm A B (inj₁ x) = inj₂ x
⊎-comm A B (inj₂ y) = inj₁ y
```

As a more involved example, a proof of

$$(A \vee \neg A) \Rightarrow \neg\neg A \Rightarrow A$$

is, following the proof of theorem 2.5.1.1,

```
lem-raa : {A : Set} → A ⊎ ¬ A → ¬ (¬ A) → A
lem-raa (inj₁ a)  k = a
lem-raa (inj₂ a') k = ⊥-elim (k a')
```

The induction principle is

```
⊎-rec : {A B : Set} → (P : A ⊎ B → Set) →
        ((x : A) → P (inj₁ x)) → ((y : B) → P (inj₂ y)) →
        (u : A ⊎ B) → P u
⊎-rec P P₁ P₂ (inj₁ x) = P₁ x
⊎-rec P P₁ P₂ (inj₂ y) = P₂ y
```

The two constructors correspond to the two introduction rules

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \; (\vee_{\mathrm{I}}^{\mathrm{l}}) \qquad\qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \; (\vee_{\mathrm{I}}^{\mathrm{r}})$$

and the non-dependent variant of the induction principle to the elimination rule

$$\frac{\Gamma, A \vdash P \qquad \Gamma, B \vdash P \qquad \Gamma \vdash A \vee B}{\Gamma \vdash P} \; (\vee_{\mathrm{E}})$$

Decidable types. A type A is *decidable* when we know whether it is inhabited or not, i.e. we have a proof of $A \vee \neg A$. We could thus define the predicate

```
Dec : Set → Set
Dec A = A ⊎ ¬ A
```

A proof of `Dec A` is a proof that `A` is decidable: by definition of the disjunction, it is either of the form `inj₁ p`, where `p` is a proof of `A`, or `inj₂ q`, where `q` is a proof of `¬ A`. Agda people like to write `yes` (resp. `no`) instead of `inj₁` (resp. `inj₂`), because it answers the question: is A provable? In the standard library, the above type is thus actually defined in the module `Relation.Nullary` as follows:

```
data Dec (A : Set) : Set where
  yes :   A → Dec A
  no  : ¬ A → Dec A
```

Since the logic of Agda is intuitionistic, the formula $A \lor \neg A$ is not provable for any type A, and not every type is decidable. However, it can be proved that no type is not decidable, see section 2.3.5:

```
nndec : (A : Set) → ¬ (¬ (Dec A))
nndec A n = n (no (λ a → n (yes a)))
```

This is further discussed in section 6.6.8.

6.5.7 Π-types. A defining feature of Agda is that it uses dependent types: a type can depend on a term. As we will see, some of the connectives admit dependent generalizations. The first one is the generalization of function types

$$A → B$$

to dependent function types

$$(x : A) → B$$

where x might occur in B. These model functions where the type B of the returned value depends on the argument x. A typical example is the replicate function of section 6.4.7, which takes a natural number n as argument and returns a vector of length n. Its type is the dependent function type

```
replicate : {A : Set} → A → (n : ℕ) → Vec A n
```

Dependent function types are also called Π-*types*, and often written

$$\Pi(x : A).B$$

instead of using the above notation. Although there is a built-in notation in Agda, one can define an inductive type for those by

```
data Π (A : Set) (B : A → Set) : Set where
  Λ : ((a : A) → B a) → Π A B
```

Namely, an element of the Π type Π A B is simply a dependent function

$$(x : A) → B x$$

From a logical point of view, it corresponds to a universal quantification which is *bounded* (we specify the type A over which the variable ranges): the above type corresponds to the logical formula

$$\forall x \in A.B(x)$$

and proof of such a formula corresponds to a function which to every x in A associates a proof of $B(x)$. This is why Agda also allows the notation

$$\forall \ x → B \ x$$

for the above type, if one is inclined to leave A implicit.

Exercise 6.5.7.1. Show that the type

$$\Pi \ \text{Bool} \ (\lambda \ \{ \ \text{false} → A \ ; \ \text{true} → B \ \})$$

is isomorphic to A × B.

6.5.8 Σ-types. Σ-*types* are a dependent variant of product types, whose elements are of the form `a , b` where `a` is of type `A` and `b` is of type `B a`: the type of the second component depends on the first component. They are defined in `Data.Product` by

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (a : A) → B a → Σ A B
```

(for technical reasons the actual definition in Agda is done using a record, but is equivalent to the above one). As for usual products, we can define two projections by

```
proj₁ : {A : Set} {B : A → Set} → Σ A B → A
proj₁ (a , b) = a
```

and

```
proj₂ : {A : Set} {B : A → Set} → (s : Σ A B) → B (proj₁ s)
proj₂ (a , b) = b
```

Again, in the second projection, note that the returned type depends on the first component.

Logical interpretation. From a logical point of view, the type

$$\text{Σ A B}$$

can be read as a bounded existential quantification and corresponds to what one would usually write

$$\exists x \in A.B(x)$$

A proof of such a formula is a pair consisting of an element x of A and a proof that x satisfies $B(x)$. In a set theoretic interpretation, it corresponds to constructing sets by comprehension, i.e. the set of elements x of A such that $B(x)$ is satisfied, what we would usually write

$$\{x \in A \mid B(x)\}$$

For instance, in set theory, given a function $f : A \to B$ (from a set A to a set B), its *image* $\text{Im}(f)$ is the subset of B consisting of elements in the image of f. It is formally defined as

$$\text{Im}(f) = \{y \in B \mid \exists x \in A.f(x) = y\}$$

This immediately translates as a definition in Agda, with two Σ types (one for the comprehension and one for the universal quantification):

```
Im : {A B : Set} (f : A → B) → Set
Im {A} {B} f = Σ B (λ y → Σ A (λ x → f x ≡ y))
```

and one can for instance show that every function $f : A \to B$ has a right inverse (or *section*) $g : \text{Im}(f) \to A$:

```
sec : {A B : Set} (f : A → B) → Im f → A
sec f (y , x , p) = x
```

The axiom of choice. In a similar vein, the *axiom of choice* states that for every relation $R \subseteq A \times B$ satisfying

$$\forall x \in A.\exists y \in B.(x, y) \in R \tag{6.1}$$

there is a function $f : A \to B$ such that

$$\forall x \in A.(x, f(x)) \in R$$

In section 6.5.9 below, we define a type `Rel A B` corresponding to relations between two types `A` and `B`, from which we can easily write a type corresponding to the axiom of choice. What might be a surprise to you is that this axiom is actually provable in Agda:

```
AC : {A B : Set} (R : Rel A B) →
     ((x : A) → Σ B (λ y → R x y)) →
     Σ (A → B) (λ f → ∀ x → R x (f x))
AC R f = (λ x → proj₁ (f x)) , (λ x → proj₂ (f x))
```

The reason is that the argument, which corresponds to the proof of (6.1), is constructive: it is a function which to every element x of type A associates a pair consisting of an element y of B and a proof that (x, y) belongs to the relation. By projecting it on the first component, we thus obtain the function f we are looking for (associating an element of B to each element of A), and we can use the second component to prove that it satisfies the required property.

However, what people have in mind when thinking of the axiom of choice: they rather have a "classical" variant where we do not have access to the proof of (6.1), but we only know its existence. A more reasonable description is thus the following formalization of the axiom of choice, where the double negation has killed the contents of the proof, see section 2.5.9:

```
postulate CAC : {A B : Set} (R : Rel A B) →
                ¬ ¬ ((x : A) → Σ B (λ y → R x y)) →
                ¬ ¬ Σ (A → B) (λ f → ∀ x → R x (f x))
```

This is discussed in further details in section 9.3.4.

6.5.9 Predicates. Predicates can be expressed in Agda; we will discuss this now.

Truth values. In classical logic, the set $\mathbb{B}$ of booleans is the set of *truth values*, i.e. the values in which we evaluate predicates: a predicate on a set A can either be false or true, and can thus be modeled as a function $A \to \mathbb{B}$. In Agda, we use intuitionistic logic and therefore we are not so much interested in whether a predicate is true or not, but rather in its proofs, so that the role of truth values is now played by `Set`. A *predicate* `P` on a type `A` can thus be seen as a term of type

```
A → Set
```

which to every element `x` of `A` associates the type of proofs of `P x`.

Relations. In classical mathematics, a *relation* R on a set A is a subset of $A \times A$, see also appendix A.1. An element x of A is said to be in relation with an element y when $(x, y) \in R$. A relation on A can also be encoded as a function

$$A \times A \to \mathbb{B}$$

or, equivalently by currying, as a function

$$A \to A \to \mathbb{B}$$

In this representation, x is in relation with y when $R(x, y) = 1$.

In intuitionistic type theory, we can describe the type of relations between two types A and B as the following type `Rel A`, obtained by replacing the set $\mathbb{B}$ of truth values with `Set` in the above description:

```
Rel : Set → Set₁
Rel A = A → A → Set
```

(this definition can be found in `Relation.Binary` in the standard library). For instance, the usual order relation `_≤_` on natural numbers (see below) can be given the type `Rel ℕ`, and equality relation `_≡_` on a type A (see section 6.6) can be given the type `Rel A`.

Inductive predicates. In Agda, we can define types inductively, and these types can depend on other types (inductive types can have parameters and indices). This means that we can define predicates by induction! For instance, the predicate on natural numbers of being even can be defined by induction by

```
data isEven : ℕ → Set where
  even-z : isEven zero
  even-s : {n : ℕ} → isEven n → isEven (suc (suc n))
```

We inductively state that 0 is even, and that if n is even then $n + 2$ is even. In other words, this corresponds to the definition of the set $E \subseteq \mathbb{N}$ of even numbers as the smallest set of numbers such that $0 \in E$ and $n \in E \Rightarrow n + 2 \in E$.

Similarly, the order relation on natural numbers can be defined with the following inductive type:

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : {n : ℕ} → zero ≤ n
  s≤s : {m n : ℕ} (m≤n : m ≤ n) → suc m ≤ suc n
```

This states that it is the smallest relation on natural numbers such that $0 \leqslant 0$, and $m \leqslant n$ implies $m+1 \leqslant n+1$. One of the main interest of defining predicates or relations inductively is of course that we can then reason by induction over those. For instance, we can show that the order relation is reflexive

```
≤-refl : {n : ℕ} → (n ≤ n)
≤-refl {zero}  = z≤n
≤-refl {suc n} = s≤s ≤-refl
```

and transitive

```
≤-trans : {m n p : ℕ} → (m ≤ n) → (n ≤ p) → (m ≤ p)
≤-trans z≤n       n≤p       = z≤n
≤-trans (s≤s m≤n) (s≤s n≤p) = s≤s (≤-trans m≤n n≤p)
```

Because of the support in Agda for reasoning by induction (and dependent pattern matching), this is often the best choice of style for defining predicates, leading to the simplest proofs, although there are many other possibilities. In order to illustrate this, the order on natural numbers could have been defined by

```
_≤_ : ℕ → ℕ → Set
m ≤ n = Σ ℕ (λ m' → m + m' ≡ n)
```

which is base on the classical equivalence, for $m, n \in \mathbb{N}$,

$$m \leqslant n \Leftrightarrow \exists m' \in \mathbb{N}.m + m' = n$$

We could also have defined it by

```
le : ℕ → ℕ → Bool
le zero n = true
le (suc m) zero = false
le (suc m) (suc n) = le m n

_≤_ : ℕ → ℕ → Set
m ≤ n = le m n ≡ true
```

We leave as an exercise to the reader to show reflexivity and transitivity with those formalizations.

Finally, as a more involved example, the implicational fragment of intuitionistic natural deduction is formalized in section 7.2: here, the relation $\Gamma \vdash A$ between a context Γ and a type A, which is true when the sequent is provable, is defined inductively.

6.6 Equality

Even equality is defined as an inductive type in Agda. The definition is given in `Relation.Binary.PropositionalEquality` by

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

The equality is typed, in the sense that we can compare only elements of the same type `A`. Moreover, there is only one way to show that two elements are equal: it is when they are the same! Because of dependent pattern matching, we will see that it is not as dumb as it might seem at first.

6.6.1 Equality and pattern matching. As a first proof with equality, let us show that the successor function on natural numbers is injective. In other words, for every natural numbers m and n, we have

$$m + 1 = n + 1 \Rightarrow m = n$$

This can be formalized as follows:

```
suc-injective : {m n : ℕ} → suc m ≡ suc n → m ≡ n
suc-injective refl = refl
```

In order to understand how such a proof works, let us study this proof step by step and reveal the implicit arguments m and n. We start with

```
suc-injective : {m n : ℕ} → suc m ≡ suc n → m ≡ n
suc-injective {m} {n} p = ?
```

By pattern matching on p (using the shortcut C-c C-c), the proof is transformed into

```
suc-injective : {m n : ℕ} → suc m ≡ suc n → m ≡ n
suc-injective {m} {.m} refl = ?
```

In order to do this, Agda uses the fact that p can only be the constructor refl, but it also knows that, in this case, the variable m must be equal to n. This explains the .m for the second optional argument: it means that it is not really an argument but something which has to be equal to m. We are thus left proving m ≡ m, and we can conclude by using refl. Most proofs involving equality are either performed in this way or by using the main properties of equality shown in next section.

6.6.2 Main properties. Apart from reflexivity, which is ensured by the constructor refl, equality can be shown to be a congruence: it is symmetric, transitive and compatible with every operation.

```
sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl
```

```
trans : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

```
cong : {A B : Set} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
cong f refl = refl
```

Two other important operations on equality are *substitutivity* which allows to transport the elements of a type along an equality

```
subst : {A : Set} (P : A → Set) → {x y : A} → x ≡ y → P x → P y
subst P refl p = p
```

and *coercion* which allow to convert an element of a type to another equal type

```
coe : {A B : Set} → A ≡ B → A → B
coe p x = subst (λ A → A) p x
```

The properties of equality will be discussed again in section 9.1.

6.6.3 Half of even numbers. As an application of the above properties, let us formalize the fact that every even number has a half, following the proof strategy presented in section 2.3. In traditional logical notation, we have to show

$$\forall n \in \mathbb{N}. \text{isEven}(n) \Rightarrow \exists m \in \mathbb{N}. m + m = n$$

The predicate isEven, which indicates whether a natural number is even or not, was already defined in section 6.5.9 and we can thus formalize our property as follows

```
even-half : {n : ℕ} → isEven n → Σ ℕ (λ m → m + m ≡ n)
even-half even-z = zero , refl
even-half (even-s e) with even-half e
even-half (even-s e) | m , p =
  suc m , cong suc (trans (+-suc m m) (cong suc p))
```

In the second case, we have by induction a number m such that $m + m = n$, and we want to construct a half for $n + 2$: this half will be $m + 1$, and we can show that it is a half using the following reasoning

$$
\begin{aligned}
(m+1) + (m+1) &= (m + (m+1)) + 1 \qquad &\text{by definition of addition,} \\
&= ((m+m) + 1) + 1 \qquad &\text{by the lemma below,} \\
&= (n+1) + 1 \qquad &\text{since } m + m = n.
\end{aligned}
$$

This can be implemented using the transitivity of equality (trans) as well as the fact that it is a congruence (we use cong suc to deduce $m + 1 = n + 1$ from $m = n$). We also use, as an auxiliary lemma, the fact that

$$
m + (n+1) = (m+n) + 1
$$

which can be shown by induction on m as follows:

```
+-suc : (m n : ℕ) → m + suc n ≡ suc (m + n)
+-suc zero    n = refl
+-suc (suc m) n = cong suc (+-suc m n)
```

6.6.4 Reasoning. The above handling of equality can be hard to track or read. A more natural way of presenting proofs can be achieved by using the ≡-Reasoning module, which displays equality in way closer to the usual one in mathematics. These helper functions can be accessed with

```
open ≡-Reasoning
```

Then, one can write a proof of $t_0 ≡ t_n$ in the form

```
begin t₀ ≡⟨ P₁ ⟩ t₁ ≡⟨ P₂ ⟩ ... ≡⟨ Pₙ ⟩ tₙ ∎
```

where P_i is a proof of $t_{i-1} ≡ t_i$. For instance, a proof of the commutativity of addition over natural numbers using this technique is

```
+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm m zero = +-zero m
+-comm m (suc n) =
  begin
    (m + suc n) ≡⟨ +-suc m n ⟩
    suc (m + n) ≡⟨ cong suc (+-comm m n) ⟩
    suc (n + m) ∎
```

The second case directly mimics the usual mathematical proof

$$m + (n+1) = (m+n) + 1 \qquad \text{by } \texttt{+-suc},$$
$$= (n+m) + 1 \qquad \text{by induction hypothesis.}$$

For comparison, a direct proof of this fact, using the properties of equality of section 6.6.2, would have been

```
+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm m zero    = +-zero m
+-comm m (suc n) = trans (+-suc m n) (cong suc (+-comm m n))
```

As usual in Agda, these notations are not built-in but defined in the standard library by

```
begin_ : {A : Set} {x y : A} → x ≡ y → x ≡ y
begin_ x≡y = x≡y


_≡⟨_⟩_ : {A : Set} (x {y z} : A) → x ≡ y → y ≡ z → x ≡ z
_ ≡⟨ x≡y ⟩ y≡z = trans x≡y y≡z


_∎ : {A : Set} (x : A) → x ≡ x
_∎ _ = refl
```

6.6.5 Definitional equality. In Agda, two terms which are convertible (i.e. reduce to a common term) are considered to be "equal". The equality we are referring to here is not ≡, but the equality which is internal to Agda, sometimes referred to as *definitional equality*: one cannot distinguish between two definitionally equal terms. For instance, over natural numbers, the term `zero + n` is definitionally equal to `n`, because this is the way we defined addition. Of course, definitional equality implies equality by `refl`:

```
+-zero' : (n : ℕ) → zero + n ≡ n
+-zero' n = refl
```

On the other side, the terms `n + zero` and `n` are not definitionally equal (there is nothing in the definition of addition which immediately allows to conclude that). The equality between these two terms can of course be proved, but requires some more work:

```
+-zero : (n : ℕ) → n + zero ≡ n
+-zero zero    = refl
+-zero (suc n) = cong suc (+-zero n)
```

Because of this, subtle variations in the definitions, even though they axiomatize isomorphic structures, can have a large impact on the length of the proofs, and one should take care of choosing the "best definition" for a concept, which requires some practice. For instance, for properties involving multiple natural numbers, the choice of the one on which we perform the induction can drastically change the size of the proof.

6.6.6 More properties with equality. Having introduced the notion of equality, we show here some more examples of properties involving it, for natural numbers and lists.

Natural numbers. We can show that zero is not the successor of any natural number (which is one of the axioms of Presburger and Peano arithmetic, see section 5.2.4), by a direct use of pattern matching:

```
zero-suc : {n : ℕ} → zero ≡ suc n → ⊥
zero-suc ()
```

Namely, when matching on the argument of type `zero ≡ suc n`, Agda knows that there can be no proof of such a type because `zero` and `suc n` do not begin with the same constructor. We can thus use the empty pattern `()` to indicate that the pattern matching contains no cases to handle. This behavior is detailed in section 8.4.5.

We can show that addition is associative by a simple induction:

```
+-assoc : (m n o : ℕ) → (m + n) + o ≡ m + (n + o)
+-assoc zero    n o = refl
+-assoc (suc m) n o = cong suc (+-assoc m n o)
```

Showing that multiplication is associative follows the same pattern, but requires some algebraic reasoning

```
*-assoc : (m n o : ℕ) → (m * n) * o ≡ m * (n * o)
*-assoc zero    n o = refl
*-assoc (suc m) n o = begin
  (m * n + n) * o     ≡⟨ *-+-dist-r (m * n) n o ⟩
  m * n * o + n * o   ≡⟨ cong (λ m → m + n * o) (*-assoc m n o) ⟩
  m * (n * o) + n * o ∎
```

where we use the fact that multiplication distributes over the addition:

```
*-+-dist-r : (m n o : ℕ) → (m + n) * o ≡ m * o + n * o
*-+-dist-r zero    n o = refl
*-+-dist-r (suc m) n o = begin
  (m + n) * o + o     ≡⟨ cong (λ n → n + o) (*-+-dist-r m n o) ⟩
  (m * o + n * o) + o ≡⟨ +-assoc (m * o) (n * o) o ⟩
  m * o + (n * o + o) ≡⟨ cong (λ n → m * o + n) (+-comm (n * o) o) ⟩
  m * o + (o + n * o) ≡⟨ sym (+-assoc (m * o) o (n * o)) ⟩
  m * o + o + n * o   ∎
```

Lists. Concatenation of lists satisfies similar properties to addition of natural numbers (after all, the type `ℕ` of natural numbers is isomorphic to the type `List` `⊤` of lists whose elements are all `tt`). Namely, we can show that the empty list is a neutral element for concatenation, on the left

```
++-empty' : {A : Set} → (l : List A) → [] ++ l ≡ l
++-empty' l = refl
```

and on the right

```
++-empty : {A : Set} → (l : List A) → l ++ [] ≡ l
++-empty []      = refl
++-empty (x :: l) = cong (λ l → x :: l) (++-empty l)
```

and that concatenation is associative

```
++-assoc : {A : Set} → (l l' l'' : List A) →
           ((l ++ l') ++ l'') ≡ (l ++ (l' ++ l''))
++-assoc []      l' l'' = refl
++-assoc (x :: l) l' l'' = cong (λ l → x :: l) (++-assoc l l' l'')
```

However, contrarily to addition, concatenation is not commutative. To wit, if it was then the concatenation of the lists [1] and [2] in both orders would be the same, which would mean that the lists [1, 2] and [2, 1] would be the same, which we know they are not. This reasoning can be formalized in Agda as follows:

```
++-not-comm :
  ¬ ({A : Set} → (l l' : List A) → (l ++ l') ≡ (l' ++ l))
++-not-comm f with f (1 :: []) (2 :: [])
++-not-comm f | ()
```

We can also show that the concatenation of two lists produces a list whose length is the sum of the lengths of the original lists:

```
++-length : {A : Set} → (l l' : List A) →
           length (l ++ l') ≡ length l + length l'
++-length []      l' = refl
++-length (x :: l) l' = cong suc (++-length l l')
```

Finally, let us present an all-time classic. We can define a function rev which reverses the order of the elements of a list: we show that applying this function twice to a list gets us back to the original list. We begin by introducing a function snoc (this is cons backwards) which adds an element at the end of a list:

```
snoc : {A : Set} → List A → A → List A
snoc []      x = x :: []
snoc (y :: l) x = y :: (snoc l x)
```

We can then define the reversion function by adding all the elements of a list at the end of the empty list:

```
rev : {A : Set} → List A → List A
rev []      = []
rev (x :: l) = snoc (rev l) x
```

We can then show that applying this function twice does not change the list given in the argument:

```
rev-rev : {A : Set} → (l : List A) → rev (rev l) ≡ l
rev-rev []      = refl
rev-rev (x :: l) =
  trans (rev-snoc (rev l) x) (cong (λ l → x :: l) (rev-rev l))
```

This proof requires to first show the following auxiliary lemma, stating that reversing a list l with x as last element will produce a list with x as first element, followed by the reversal of the rest of the list:

```
rev-snoc : {A : Set} → (l : List A) → (x : A) →
           rev (snoc l x) ≡ x :: (rev l)
rev-snoc []      x = refl
rev-snoc (y :: l) x = cong (λ l → snoc l y) (rev-snoc l x)
```

6.6.7 The J rule. If we define equality as

```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x
```

the associated induction principle is called the *J rule*:

```
J : {A : Set} {x y : A} (p : x ≡ y)
    (P : (x y : A) → x ≡ y → Set)
    (r : (x : A) → P x x refl)
    → P x y p
J {A} {x} {.x} refl P r = r x
```

It reads as follows: in order to prove a property P depending on a proof p of equality between two elements x and y, it is enough to prove it when this proof is refl.

In practice, we have seen in section 6.6 that the definition usually taken in Agda is slightly different (it uses an parameter instead of an index for the first argument of type A), so that the resulting induction principle is a variant on the above one:

```
J : {A : Set} (x : A) (P : (y : A) → x ≡ y → Set)
    (r : P x refl) (y : A) (p : x ≡ y) → P y p
J x P r .x refl = r
```

6.6.8 Decidable equality. Recall from section 6.5.6 that a type A is decidable when either A or $\neg A$ is provable, and we write Dec A for the type of proofs of decidability of A: such a proof is either yes p, where p is a proof of A, or no q, where q is a proof of $\neg A$. A relation on a type A is *decidable* when the type R x y is decidable for every elements x and y of type A. The standard library defines, in the module Relation.Binary, the following predicate:

```
Decidable : {A : Set} (R : A → A → Set) → Set
Decidable {A} R = (x y : A) → Dec (R x y)
```

A term of type Decidable R is a proof that the relation R is decidable.

A type A has *decidable equality* when the equality relation _≡_ on A is decidable (some people also say that A is *discrete* in this situation). This means that we have a function (i.e. an algorithm) which is able to determine, given two elements of A, whether they are equal or not. To be precise, we not only have the information of whether they are equal or not, which would be a boolean, but actually a proof of their equality or a proof of their inequality (see section 8.4.5 for a use of this).

Equality on any finite type is always decidable. For instance, in the case of booleans:

```
_≟_ : Decidable {A = Bool} _≡_
false ≟ false = yes refl
false ≟ true  = no (λ ())
true  ≟ false = no (λ ())
true  ≟ true  = yes refl
```

The type of natural numbers also has decidable equality:

```
_≟_ : Decidable {A = ℕ} _≡_
zero  ≟ zero  = yes refl
zero  ≟ suc  n = no (λ ())
suc m ≟ zero   = no (λ ())
suc m ≟ suc  n with m ≟ n
suc m ≟ suc .m | yes refl = yes refl
suc m ≟ suc  n | no  ¬p   = no  (λ p → ¬p (suc-injective p))
```

However, we do not expect that the equality is decidable on the type $\mathbb{N} \to \mathbb{N}$. One reason is that our reasoning techniques are very limited on functions, in particular we cannot perform pattern matching on functions, and thus cannot perform a proof in the same spirit as above. The other reason is that, intuitively, two functions f and g on natural numbers are equal when they $f(n) \equiv g(n)$ for every natural number n (this is not always true though, see section 9.1.5), and we do not expect that there is an algorithm which will be able to compare all the images of f and g on every natural number n in finite time...

6.6.9 Heterogeneous equality. We would finally like to present a variant of equality due to McBride [McB00], called *heterogeneous equality*, which allows to compare (seemingly) distinct types. In order to understand its use, let us try to show that the concatenation of vectors is associative. Given three vectors l, l' and l", with elements of type A, of respective lengths m, n and o, we thus want to show

$$(l \mathbin{++} l') \mathbin{++} l" \equiv l \mathbin{++} (l' \mathbin{++} l")$$

...except that this expression does not make sense! Namely, the equality ≡ can only be used to compare terms of the same type and, here, this is not the case: the left and right sides respectively have types

```
Vec A ((m + n) + o)          Vec A (m + (n + o))
```

which are not the same. Of course, the two types are propositionally equal: we can prove

```
Vec A ((m + n) + o) ≡ Vec A (m + (n + o))
```

by

```
cong (Vec A) (+-assoc m n o)
```

But the two types are not definitionally equal, which is what is required in order to compare terms with ≡.

Proof with standard equality. In order to perform our comparison, we can use `coe` and the above propositional equality in order to cast one of the members to have the same type as the other one. Namely, the term

```
coe (cong (Vec A) (+-assoc m n o))
```

has type

```
Vec A ((m + n) + o) → Vec A (m + (n + o))
```

and we can use it to "cast" `(l ++ l') ++ l''` in order to change its type to the same one as `l ++ (l' ++ l'')`, after which we can compare the two with ≡. We can finally prove associativity of concatenation of vectors as follows:

```
++-assoc : {A : Set} {m n o : ℕ} →
  (l : Vec A m) → (l' : Vec A n) → (l'' : Vec A o) →
  coe (cong (Vec A) (+-assoc m n o))
    ((l ++ l') ++ l'') ≡ l ++ (l' ++ l'')
++-assoc [] l' l'' = refl
++-assoc {_} {suc m} {n} {o} (x ∷ l) l' l'' =
  ∷-cong x (+-assoc m n o) (++-assoc l l' l'')
```

The above proof uses the following auxiliary lemma which states that if `l` and `l'` are propositionally equal vectors, up to propositional equality of their types as above, then `x ∷ l` and `x ∷ l'` are also propositionally equal:

```
∷-cong : {A : Set} → {m n : ℕ} {l : Vec A m} {l' : Vec A n} →
     (x : A) → (p : m ≡ n) → coe (cong (Vec A) p) l ≡ l' →
     coe (cong (Vec A) (cong suc p)) (x ∷ l) ≡ x ∷ l'
∷-cong x refl refl = refl
```

As you can observe, the statement of those properties is considerably obscured by the use of `coe`, which is used to coerce the type of terms so that they can be compared to other terms, as explained above.

Proof with heterogeneous equality. In order to overcome this problem, we can use the *heterogeneous equality* relation, also sometimes called *John Major equality*, which is defined by

```
data _≅_ : {A B : Set} (x : A) (y : B) → Set where
  refl : {A : Set} {x : A} → x ≅ x
```

in the module `Relation.Binary.HeterogeneousEquality`. It is a variant of propositional equality, which allows comparing two elements `x` and `y` of distinct types `A` and `B`. It is however a reasonable notion of equality because the constructor `refl` only allows to construct an heterogeneous equality when `A` and `B` are the same. This ability of comparing elements of distinct types allows formulating and proving the associativity of vectors in a much easier way:

```
++-assoc : {A : Set} {m n o : ℕ}
           (l : Vec A m) (l' : Vec A n) (l'' : Vec A o) →
           ((l ++ l') ++ l'') ≅ (l ++ (l' ++ l''))
++-assoc [] l' l'' = refl
++-assoc {_} {suc m} {n} {o} (x ∷ l) l' l'' =
  ∷-cong x (+-assoc m n o) (++-assoc l l' l'')
```

with the preliminary lemma

```
::-cong : {A : Set} {m n : ℕ} {l : Vec A m} {l' : Vec A n} →
         (x : A) → m ≡ n → l ≅ l' → x :: l ≅ x :: l'
::-cong x refl refl = refl
```

The reader should be warned that heterogeneous equality is not entirely satisfactory. Firstly, if x and y are two elements of the same type A, we cannot formally show that x ≅ y implies x ≡ y (unless we assume axiom K, see section 9.1.6). Secondly, being able to compare elements of any two types A and B seems quite worrying, the only thing we really need here is to compare elements of such types when A ≡ B: a more satisfactory definition is given in section 9.5.2.

6.7 Proving programs in practice

We shall now briefly explain and illustrate how we can prove a program is correct. Of course, there is no universally accepted notion of what we mean by the *correctness* of a program: it only means that it agrees with a *specification*, which can usually be expressed as a logical formula, and whose definition is left to the person certifying the program. We can however classify correctness properties in three rough families.

- *Absence of errors*: the program always uses functions with arguments in the domain where functions are supposed to operate correctly. For instance, we want to avoid dividing by zero or dereferencing null pointers.

- *Invariants*: we show that some properties are always satisfied during the execution of the program, e.g. the variable x will always contain a strictly positive number.

- *Functional properties*: the program computes the expected output on any given input, e.g. given a natural number n, the function square n will produce a natural number m such that $m = n^2$.

We have ordered them from the less to the most precise: the first kind only ensures that basic programming errors are avoided, the second one that our program satisfies some good properties, and the last one that it fully behaves as expected. Of course, these are not disjoint: absence of errors is a particular kind of invariant, and proving functional properties usually requires showing invariants first.

6.7.1 Extrinsic vs intrinsic proofs. There are two common approaches in order to prove properties of programs. The *extrinsic* approach is the way one traditionally proceeds [CLRS09]: we first write our program and then we prove properties about it. The *intrinsic* approach is often more adapted to proving programs in dependent type theory: it consists in changing the type of our program, so that this type incorporates the properties we want to prove (remember that we can see any reasonable formula as a type!). For instance, suppose that we want to show that our sorting algorithm sorts lists of natural numbers.

- In the *extrinsic* approach we implement our algorithm as a function sort whose type is

```
List ℕ → List ℕ
```

as usual and then show that this function actually sorts a list, i.e. prove the proposition

```
(l : List ℕ) → sorted (sort l)
```

where `sorted l` is a predicate stating that a list `l` is sorted.

– In the *intrinsic* approach, we directly implement our algorithm as a function `sort` of type

```
List ℕ → SortedList
```

where `SortedList` is the type of *sorted* lists of natural numbers.

This example is detailed in section 6.7.2 for the insertion sort algorithm. The intrinsic approach usually results in shorter code, and is not significantly harder than the extrinsic one, although it usually requires more thought in order to formulate the property we want to prove in a way which will give rise to an elegant proof.

Length and concatenation. As a simple example, suppose that we want to show that the the length of the concatenation of two lists is the sum of their lengths. In the extrinsic approach, we would define concatenation as usual, see section 6.4.5:

```
_++_ : {A : Set} → List A → List A → List A
[]       ++ l' = l'
(x :: l) ++ l' = x :: (l ++ l')
```

and then shows that it is additive with respect to lengths, see section 6.6.6:

```
++-length : {A : Set} → (l l' : List A) →
            length (l ++ l') ≡ length l + length l'
++-length []       l' = refl
++-length (x :: l) l' = cong suc (++-length l l')
```

In the intrinsic approach, we would consider the type of lists of a given length, i.e. the type of vectors, and give the following type to the concatenation, see section 6.4.7:

```
_++_ : {m n : ℕ} {A : Set} → Vec A m → Vec A n → Vec A (m + n)
[]       ++ l  = l
(x :: l) ++ l' = x :: (l ++ l')
```

which both defines the concatenation and shows the property we were looking for at once.

6.7.2 Insertion sort. As a more involved example, we consider the insertion sort algorithm to sort a list. We recall that a list $l = [x_1, x_2, \ldots, x_n]$ is *sorted* when $x_1 \leqslant x_2 \leqslant \ldots \leqslant x_n$. For simplicity, we consider here lists of natural numbers, which are compared using to the usual total order. Given a list l, and an element x, we can insert the element x into l, in order to obtain a sorted list, by comparing x with the elements of l from left to right and inserting it before the first element which is greater than it. Formally, we can define a function $\mathrm{insert}(x, l)$ by recursion on l by

$$\mathrm{insert}(x, []) = [x]$$

$$\mathrm{insert}(x, y :: l) = \begin{cases} x :: y :: l & \text{if } x \leqslant y, \\ y :: \mathrm{insert}(x, l) & \text{otherwise.} \end{cases}$$

The insertion sort algorithm then proceeds, in order to sort a given list, by iteratively inserting all its elements in a list which is initially the empty list. We write $\mathrm{sort}(l)$ for the list obtained in this way:

$$\mathrm{sort}([]) = []$$
$$\mathrm{sort}(x :: l) = \mathrm{insert}(x, \mathrm{sort}(l))$$

If you prefer OCaml code:

```
let rec insert x = function
  | [] -> [x]
  | y::l ->
    if x <= y then x::y::l
    else y::(insert x l)

let rec sort = function
  | [] -> []
  | x::l -> insert x (sort l)
```

In order to prove the functional correctness of our algorithm, we have to show that, given any list as input, the output is a sorted list. It can be shown that

– the empty list is sorted,

– given a sorted list l and any element x, the list $\mathrm{insert}(x, l)$ is sorted,

from which we deduce, by induction on l, that the list $\mathrm{sort}(l)$ is sorted for any list l, see [CLRS09, section 2.1].

Extrinsic approach. The correctness of insertion sort using the extrinsic approach is shown in figure 6.2. We can define the function `insert`, to insert an element in a list, and `sort`, to sort a list, by a direct translation of the above definitions (for simplicity, we only handle the case of lists of natural numbers). Note that the sorting function has the usual type

```
sort : List ℕ → List ℕ
```

In the definition of the insertion function, we use the predicate

```
_≤?_ : (m n : ℕ) → Dec (m ≤ n)
```

```
open import Data.Product
open import Data.Unit hiding (_≤_ ; _≤?_)
open import Relation.Nullary
open import Data.Nat
open import Data.Nat.Properties
open import Data.List

insert : (x : ℕ) → (l : List ℕ) → List ℕ
insert x [] = x :: []
insert x (y :: l) with x ≤? y
insert x (y :: l) | yes _ = x :: y :: l
insert x (y :: l) | no  _ = y :: insert x l

sort : List ℕ → List ℕ
sort []      = []
sort (x :: l) = insert x (sort l)

_≤*_ : (x : ℕ) → (l : List ℕ) → Set
x ≤* []      = T
x ≤* (y :: l) = x ≤ y × x ≤* l

≤*-trans : {x y : ℕ} → (x ≤ y) → (l : List ℕ) → y ≤* l → x ≤* l
≤*-trans x≤y []        tt              = tt
≤*-trans x≤y (z :: l) (y≤z , y≤*l) = ≤-trans x≤y y≤z , ≤*-trans x≤y l y≤*l

≤*-insert : {x y : ℕ} → (x ≤ y) → (l : List ℕ) →
            x ≤* l → x ≤* (insert y l)
≤*-insert x≤y [] tt = x≤y , tt
≤*-insert {x} {y} x≤y (z :: l) x≤*zl with y ≤? z
≤*-insert x≤y (z :: l) x≤*zl         | yes _ = x≤y , x≤*zl
≤*-insert x≤y (z :: l) (x≤z , x≤*l) | no  _ = x≤z , ≤*-insert x≤y l x≤*l

sorted : (l : List ℕ) → Set
sorted []      = T
sorted (x :: l) = x ≤* l × sorted l

insert-sorting : (x : ℕ) → (l : List ℕ) → sorted l → sorted (insert x l)
insert-sorting x [] s = tt , tt
insert-sorting x (y :: l) (y≤*l , sl) with x ≤? y
insert-sorting x (y :: l) (y≤*l , sl) | yes x≤y =
  (x≤y , (≤*-trans x≤y l y≤*l)) , (y≤*l , sl)
insert-sorting x (y :: l) (y≤*l , sl) | no x≰y =
  (≤*-insert (≰⇒≥ x≰y) l y≤*l) , insert-sorting x l sl

sorting : (l : List ℕ) → sorted (sort l)
sorting []      = tt
sorting (x :: l) = insert-sorting x (sort l) (sorting l)
```

Figure 6.2: Correctness of insertion sort (extrinsic version).

which shows that the order on natural numbers is decidable, which is proved similarly as for equality, see section 6.6.8.

Since lists are defined by induction, and all the reasoning about those will be performed by induction, it is better to define the predicate of being sorted for a list by induction. In order to do so, we first define a relation $\leqslant^*$ between natural numbers and lists such that $x \leqslant^* l$ whenever $x \leqslant y$ for every element y of l, i.e. the elements of l are bounded below by x. This is defined by induction on l by

- $x \leqslant^* l$ always holds when l is the empty list,

- $x \leqslant^* (y :: l)$ whenever $x \leqslant y$ and $x \leqslant^* l$.

We can then define the predicate of being sorted for a list by induction on the list by

- the empty list is always sorted,

- a list $x :: l$ is sorted whenever $x \leqslant^* l$ and l is sorted.

Finally, using two easy lemmas involving the relation $\leqslant^*$, we can show that given any number x and list l which is sorted, the list $\mathrm{insert}(x, l)$ is also sorted (this is `insert-sorting`), from which we can deduce that, for any list l, the list $\mathrm{sort}(l)$ is sorted (this is `sorting`).

Intrinsic approach. We show the intrinsic approach to show correctness of insertion sort in figure 6.3. Here, we give to the sorting function the type

```
sort : (l : List ℕ) → SortedList
```

which directly specifies that it returns a sorted list. Here, `SortedList` is the type of sorted lists of natural numbers, which can be defined inductively as follows: a sorted list is

- either empty, or

- of the form $x :: l$ where $x \leqslant^* l$ and l is a sorted list.

There is a subtlety, however: we now want the relation $x \leqslant^* l$ to apply to a sorted list l, so that it should be defined by mutual induction with the notion of sorted list. This kind of definition is called an inductive-inductive type, see section 8.4.3, and requires to declare the type of both `SortedList` and `_≤*_` beforehand.

The insertion function basically takes an element and a sorted list and returns the sorted list resulting from the insertion of the element. However, in order to show that the result is a sorted list, we need to return a second element which states that this result `l'` satisfies a property akin to `≤*-insert` in the extrinsic approach (you should try by yourself in order to understand why), and the type if the insertion function is

```
insert : (x : ℕ) (l : SortedList) →
         Σ SortedList (λ l' → {y : ℕ} → y ≤ x → y ≤* l → y ≤* l')
```

```
open import Data.Product
open import Data.Unit hiding (_≤_ ; _≤?_)
open import Relation.Nullary
open import Data.Nat
open import Data.Nat.Properties
open import Data.List

data SortedList : Set
data _≤*_        : ℕ → SortedList → Set

data SortedList where
  empty : SortedList
  cons  : (x : ℕ) (l : SortedList) (le : x ≤* l) → SortedList

data _≤*_ where
  ≤*-empty : {x : ℕ} → x ≤* empty
  ≤*-cons  : {x y : ℕ} {l : SortedList} →
             x ≤ y → (le : y ≤* l) → x ≤* (cons y l le)

≤*-trans : {x y : ℕ} {l : SortedList} → x ≤ y → y ≤* l → x ≤* l
≤*-trans x≤y ≤*-empty           = ≤*-empty
≤*-trans x≤y (≤*-cons y≤z z≤*l) = ≤*-cons (≤-trans x≤y y≤z) z≤*l

insert : (x : ℕ) (l : SortedList) →
         Σ SortedList (λ l' → {y : ℕ} → y ≤ x → y ≤* l → y ≤* l')
insert x empty =
  cons x empty ≤*-empty , (λ y≤x _ → ≤*-cons y≤x ≤*-empty)
insert x (cons y l y≤*l) with x ≤? y
... | yes x≤y =
  cons x (cons y l y≤*l) (≤*-cons x≤y y≤*l) ,
  (λ z≤x z≤*yl → ≤*-cons z≤x (≤*-cons x≤y y≤*l))
... | no  x≰y with insert x l
...              | l' , p =
  (cons y l' (p (≰⇒≥ x≰y) y≤*l)) ,
  ((λ { z≤x (≤*-cons z≤y _) → ≤*-cons z≤y (p (≰⇒≥ x≰y) y≤*l) }))

sort : (l : List ℕ) → SortedList
sort []       = empty
sort (x ∷ l) = proj₁ (insert x (sort l))
```

Figure 6.3: Correctness of insertion sort (intrinsic version).

6.7.3 The importance of the specification. Once we have performed such a proof, does this guarantee that our function is correct? Well... yes and no! On the bright side, our rigorous proof does indeed guarantee that the sorting function will always return a sorted list, whichever list we provide to it as input. This is actually true for eternity.

However, one might be surprised to find out the following function also has the same type:

```
bad : (l : List ℕ) → SortedList
bad l = empty
```

This function always returns the empty list, whichever list is provided as input. This will not usually be considered as a valid sorting functions, although it fills the bill. The empty list is, after all, a sorted list. The culprit is not the proof assistant nor the proof here, but the *specification* itself: what we expect from a sorting function is not only to return a sorted list, but also that the returned list has the same elements as the one given as argument.

Exercise 6.7.3.1. Show that the insertion sort function satisfies the strengthened specification.

This kind of problem is not purely theoretical: in an earlier version of this book, the function given in figure 6.3 was actually wrong, and this remained unnoticed because the specification of sorted lists was also wrong...

6.8 Termination

6.8.1 Termination and consistency. In order to maintain consistency, Agda ensures that all the defined functions are *terminating*, by which we mean that they will always give a result after a finite amount of time. To understand why this is required, we can force it to accept a non-terminating function and this is what happens (spoiler: inconsistency). This can be achieved by using the pragma `{-# TERMINATING #-}` before the definition of a function, which means "trust me, this function is terminating". For instance, the function f defined on natural numbers by $f(n) = f(n+1)$ is clearly not terminating. It can be given the type `ℕ → ⊥`, from which it is easy to make the system inconsistent:

```
{-# TERMINATING #-}
f : ℕ → ⊥
f n = f (suc n)

absurd : ⊥
absurd = f zero

0≡1 : 0 ≡ 1
0≡1 = ⊥-elim absurd
```

Yes, we have managed to prove $0 = 1$. If we do not use the pragma, Agda correctly detects that the function f is problematic and prevents us from defining it:

```
Termination checking failed for the following functions:
```

```
  f
Problematic calls:
  f (suc n)
```

6.8.2 Structural recursion. In order to ensure that the programs are terminating, Agda uses a "rough" criterion, which is simple to check and safe, in the sense that it ensures every accepted program is terminating. This criterion is that recursive programs must be *structurally recursive*, meaning that all the recursive calls must be done on strict subterms of the argument (we say that the argument is *structurally decreasing*).

For instance, the following function computes the n-th term of the Fibonacci sequence, defined by $f_0 = 0$, $f_1 = 1$ and $f_{n+2} = f_{n+1} + f_n$:

```
fib : ℕ → ℕ
fib zero        = zero
fib (suc zero)  = suc zero
fib (suc (suc n)) = fib n + fib (suc n)
```

In the third case, the argument is `suc (suc n)`, whose strict subterms are `suc n` and `n`, see section 5.1.2. Since the recursive calls are performed with those as arguments, the program is accepted. If we had instead used recursive calls of one of the following forms then the program would be rejected

- `fib (suc (suc n))`: the argument `suc (suc n)` is a subterm of itself, but not a strict one,

- `fib (zero + n)`: the term `zero + n` is not a strict subterm of `suc (suc n)`, i.e. the first does not occur in the second; as you can see the notion of subterm has to be taken purely syntactically here, no reduction is performed (the fact that `zero + n` reduces to `n` is not taken in account).

Multiple arguments. In the case where the function has two arguments (and this generalizes to multiple arguments), either the first argument must be structurally decreasing (in which case there is no restriction on the second one) or it should stay the same and the second argument must be structurally decreasing. Pairs of arguments are thus compared using the lexicographic order, see appendix A.3.3. For instance, the following Ackermann function is also accepted:

```
ack : (x y : ℕ) -> ℕ
ack zero    n        = suc n
ack (suc m) zero     = ack m (suc zero)
ack (suc m) (suc n) = ack m (ack (suc m) n)
```

In the second case, the first argument `m` is a subterm of the first argument `suc m` of the function. In the third case, one recursive call is performed with `m` as first argument, which is a subterm of the first argument `suc m` and the second recursive call is performed with `suc n` as first argument, which stays unchanged, and `n` as second argument, which is a subterm of the second argument `suc n`.

Rejecting valid programs. The restriction to structurally recursive functions has the advantage to be simple, but the downside is that some programs which are not problematic are rejected by Agda, because they do not satisfy this criterion even though they are terminating. For instance, consider the following function which computes the quotient of two natural numbers:

```
div : ℕ → ℕ → ℕ
div m n with m <? suc n
div m n | yes _ = zero
div m n | no  _ = suc (div (m ∸ suc n) n)
```

To be precise, `div m n` computes the quotient of `m` and `n + 1`, in order to avoid problematic divisions by zero. Even though it is terminating, this function is rejected. Namely, in the second case, `m ∸ suc n` is not a strict subterm of `m`: Agda is not smart enough to notice that the recursive calls are performed with strictly decreasing values for `m` and must therefore be terminating. However, this does not mean that we cannot define division in Agda: there are other ways to formulate division, which are only slightly more complicated than the usual way shown above – and they get accepted, see section 6.8.7.

6.8.3 A bit of computability. The criterion used by Agda to determine if a program is terminating is overly restrictive and it has to be so: it was shown by Turing that the *halting problem*, which consists in deciding whether a program terminates or not, is undecidable, i.e. there is no program which given a program as input determines whether it eventually terminates or not [Tur37b]. However, we have indicated that for a given function, even if the straightforward terminating implementation is abusively rejected by Agda, there is usually a way to reformulate it in order to obtain an implementation which is accepted. We show here that there are however some computable functions which cannot be implemented: the language Agda is not *Turing complete.*

Computable functions. Given two sets A and B, a (partial) *function* f from A to B, associates to some elements x of A an image $f(x)$ in B. We write $\mathrm{dom}(f)$ for the set of elements of A which have an image under f, called the *domain* of f, and say that the function f is *total* when $\mathrm{dom}(f) = A$. We say that a function

$$f : \mathbb{N} \to \mathbb{N}$$

is implemented by an OCaml function

```
f : int -> int
```

when

- for every natural number $n \in \mathrm{dom}(f)$, the computation of `f n` terminates and its result is $f(n)$,

- for every natural number $n \notin \mathrm{dom}(f)$, the computation of `f n` does not return a result.

Of course, we can define similarly the notion of an implementation of the function f in any other programming language which canonically contains the natural numbers, such as Agda, and we could extend the notion of computable

function to other data types than natural numbers. A function which can be implemented by some function in OCaml is said to be *computable* (of course, we could have chosen any other reasonable programming language in order to define computable functions, see section 3.3.6 for another possible definition).

For instance, the function $f : \mathbb{N} \to \mathbb{N}$ such that $\mathrm{dom}(f)$ is the set of odd natural numbers and $f(n) = n + 1$ for every $n \in \mathrm{dom}(f)$ can be implemented in OCaml by

```
let rec f n = if n mod 2 = 0 then f n else n + 1
```

or by

```
let rec f n = while n mod 2 = 0 do () done; n + (n mod 2)
```

and is thus computable. As illustrated above, there are generally multiple ways to implement a given function.

Programming with total functions. The programming language Agda has one particularity compared to usual programming languages: since every function is terminating, all the functions which can be implemented are total. From this follows the following property.

Theorem 6.8.3.1. In a programming language such as Agda in which all the functions which can be implemented are total, there is a total computable function which cannot be implemented.

Proof. The idea is that the if all total computable functions were implementable in Agda then some partial function would also be implementable. Here is a detailed sketch of the proof. The functions $f : \mathbb{N} \to \mathbb{N}$ which can be implemented in Agda are described by a string, and are therefore countable: we can enumerate those and write f_i for the i-th implementable function. The function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $g(i, n) = f_i(n)$ is also implementable: given an argument (i, n) the function g enumerates all strings in order and, for each string, tests whether it is a valid Agda definition of a function of type $\mathbb{N} \to \mathbb{N}$, until it finds the i-th such function f_i, at which points it returns the evaluation of f_i on the argument n (this would require programming an evaluator of Agda functions in OCaml, which can be done). Suppose that g can be implemented in Agda (otherwise, we can conclude immediately). Then the function $d : \mathbb{N} \to \mathbb{N}$ defined by $d(n) = g(n, n) + 1$ is clearly also implementable. Therefore, there is an index i such that $d = f_i$ and we have

$$d(i) = g(i, i) + 1 = f_i(i) + 1 = d(i) + 1$$

Contradiction. $\qquad \square$

Functions which cannot be implemented are rare. In practice, all the usual functions that one manipulates can be implemented in Agda. An example of a function which cannot be implemented in Agda is an interpreter for the Agda language itself.

6.8.4 The number of bits. We have mentioned that the restriction to structurally recursive functions is quite strong and rejects perfectly terminating functions. Let us study an example and see the available workarounds. We consider the function bits which associates to every natural number n, the number of bits necessary to write it in base 2. For instance,

$$\text{bits}(0) = 0 \quad \text{bits}(1) = 1 \quad \text{bits}(2) = 2 \quad \text{bits}(3) = 2 \quad \text{bits}(4) = 3 \quad \ldots$$

This function is essentially a rounded base 2 logarithm: it can be expressed as

$$\text{bits}(n) = 1 + \lfloor \log_2(n) \rfloor$$

where, by convention, $\log_2(0) = -1$. This function satisfies the following equations

$$\text{bits}(0) = 0 \qquad\qquad \text{bits}(n+1) = 1 + \text{bits}(\lfloor n/2 \rfloor)$$

which allows it to be computed recursively. In OCaml, it can thus be implemented as

```
let rec bits n =
  if n = 0 then 0 else 1 + bits (n / 2)
```

This function is terminating because the recursive call is done on a smaller natural number, thanks to the division by 2. In order to perform an analogous definition in Agda, we can define division by 2 with

```
div2 : ℕ → ℕ
div2 zero         = zero
div2 (suc zero)   = zero
div2 (suc (suc n)) = suc (div2 n)
```

and then translate the above OCaml definition as

```
bits : ℕ → ℕ
bits zero    = zero
bits (suc n) = suc (bits (div2 (suc n)))
```

This function is not accepted by Agda (without enforcing termination), because the recursive call of `bits` is performed on `div2 (suc n)`, which is not a strict subterm of the argument `suc n`.

6.8.5 The fuel technique. In order to define our function, a general technique consists in adding new arguments to it, so that the recursive calls are performed with one of these arguments being structurally decreasing. Typically, we can add as argument a natural number which will decrease at each call (when the function is called with `suc n`, the recursive call is performed with `n`), provided that we know in advance a bound on the number of recursive calls (i.e. we also have to add a proof that this argument will be non-zero so that we can decrease it). This is called the *fuel technique* because this natural number can be thought of as some fuel which we are consuming in order to perform the recursive calls.

For instance, in order to define the bits function, we can add a natural number `fuel` as argument to the function, which is to be structurally decreasing:

```
bits-fuel : (n : ℕ) → (fuel : ℕ) → ℕ
bits-fuel zero f          = zero
bits-fuel (suc n) zero    = ?
bits-fuel (suc n) (suc f) = suc (bits-fuel (div2 (suc n)) f)
```

In the case the original argument `n` is

– `zero`: we can return `zero`,

– `suc n`:

  – if the fuel is of the form `suc f`, we can make a recursive call on `div2 (suc n)` with `f` as fuel: the fuel argument is structurally decreasing (we have consumed one unit of fuel),

  – if the fuel is `zero` however, we do not know what to do (this is the `?` above): we cannot perform a recursive call with structurally smaller fuel because we do not have fuel anymore (there is no strict subterm of `zero`).

In order to overcome the problem encountered in the last case, we have to ensure that we never "run out of fuel", i.e. that the fuel is strictly positive when we need to perform a recursive call. This can be achieved by adding a second additional argument which ensures an invariant on the fuel which will enforce this. For instance, we can add the requirement that the fuel is always greater than the original argument `n`. When performing a recursive call, we will have to show that this invariant is preserved: under the hypothesis $n + 1 \leqslant f + 1$, we have to show $(n + 1)/2 \leqslant f$, which can be done as follows:

$$(n + 1)/2 \leqslant (f + 1)/2 \leqslant f$$

We thus define

```
bits-fuel : (n : ℕ) → (fuel : ℕ) → (n ≤ fuel) → ℕ
bits-fuel zero    f        p = zero
bits-fuel (suc n) zero     ()
bits-fuel (suc n) (suc f) p =
  suc (bits-fuel (div2 (suc n)) f n+1/2≤f)
  where
  n+1/2≤f : div2 (suc n) ≤ f
  n+1/2≤f = begin
    div2 (suc n) ≤⟨ ≤-div2 p ⟩
    div2 (suc f) ≤⟨ ≤-div2-suc f ⟩
    f            ∎
```

This follows the same pattern as the previous definition, except that we know that the problematic case where the original argument is `suc n` and the fuel is `zero` will not happen: by the third argument, we would have `suc n ≤ zero`, which is impossible. The code is longer than above because, when performing the recursive call on `div2 n` with `f` as fuel, we have to provide a third argument, which shows that the invariant is preserved, i.e. `div2 n ≤ f` holds. This is shown in the lemma named `n+1/2≤f`, using two auxiliary lemmas

```
≤-suc : (n : ℕ) → n ≤ suc n
```

and

```
≤-div2-suc : (n : ℕ) → div2 (suc n) ≤ n
```

whose proof is left to the reader. We can finally define the bits function by providing, as fuel, a high enough number. For instance n is suitable:

```
bits : ℕ → ℕ
bits n = bits-fuel n n ≤-refl
```

6.8.6 Well-founded induction. We now present a generalization of this technique called *well-founded induction*. The fundamental reason why the fuel technique is working is that we are decreasing some natural number and there is no infinite strictly decreasing sequence of integers so that we know that the recursive calls will stop at some point. The technique presented here axiomatizes this situation and is also detailed in appendix A.3.

Well-founded induction and recursion. In mathematics, a *relation R* on a set A is a subset of $A \times A$. Given elements x and y of A such that $(x, y) \in R$, we write $x \mathrel{R} y$, and think of x as being "smaller" than y. The relation R is *well-founded* when there is no infinite sequence $(x_i)_{i \in \mathbb{N}}$ of elements of A which is decreasing, i.e. such that $x_{i+1} \mathrel{R} x_i$ for every index i:

$$\ldots R\, x_3 \mathrel{R} x_2 \mathrel{R} x_1 \mathrel{R} x_0$$

Example 6.8.6.1. On the set $\mathbb{N}$ of natural numbers the following two relations are well-founded:

– the relation $\prec$ such that $n \prec n + 1$ for every $n \in \mathbb{N}$,

– the usual strict order relation $<$.

Example 6.8.6.2. If you are looking for counter-examples, the relation $<$ on $\mathbb{R}$ or on $\mathbb{Q}$ is not well-founded, nor is the relation $\leqslant$ on $\mathbb{N}$.

Given $x \in A$, we write

$$\downarrow x = \{y \in A \mid y \mathrel{R} x\}$$

for the set of *predecessors* of x. The following *well-founded induction* principle holds for well-founded relations, which generalizes the usual induction principle on natural numbers:

Theorem 6.8.6.3 (Well-founded induction). Suppose given a set A, a well-founded relation R on A and a predicate P on the elements of A such that, for every $x \in A$, if P holds on every element of $\downarrow x$ then P holds on x. Then P holds for every element of A.

Example 6.8.6.4. On $\mathbb{N}$, the well-founded induction principle associated to $\prec$ is the usual induction principle: given a predicate P such that $P(0)$ holds, and $P(n)$ implies $P(n + 1)$ for every $n \in \mathbb{N}$, we have that $P(n)$ holds for every $n \in \mathbb{N}$.

Example 6.8.6.5. On $\mathbb{N}$, the well-founded induction principle associated to $<$ is the strong induction principle: given a predicate P such that, for every $n \in \mathbb{N}$,

if $P(i)$ holds for every $i < n$ then $P(n)$ holds, we have that $P(n)$ holds for every $n \in \mathbb{N}$. In formulas, this induction principle can be formulated as

$$(\forall n \in \mathbb{N}.(\forall m \in \mathbb{N}.m < n \Rightarrow P(m)) \Rightarrow P(n)) \Rightarrow \forall n \in \mathbb{N}.P(n)$$

for any predicate P on natural numbers.

Given a function $f : A \to B$ and $A' \subseteq A$, we write $f|_{A'} : A' \to B$ for the function f restricted to A'. The following *well-founded recursion* principle can be shown, which generalizes the definition of a function by recursion:

Theorem 6.8.6.6 (Well-founded recursion)*.* Suppose given sets A and B, a well-founded relation R on A and function r which to every $x \in A$ and function $\downarrow x \to B$ associates an element of B. Then there is a unique function $f : A \to B$ such that, for every $x \in A$,

$$f(x) = r(x, f|_{\downarrow x})$$

In the above theorem, we are defining a function f by recursion: the function r describes how to produce the value of $f(x)$ from x and all the values of $f(y)$ for y smaller than x, i.e. such that $y \mathrel{R} x$. Note that the type of the r is the dependent type $(\Sigma(x : A).(\downarrow x \to B)) \to B$.

Example 6.8.6.7*.* Consider the well-founded relation $\prec$ on $\mathbb{N}$. We have

$$\downarrow 0 = \emptyset \qquad\qquad\qquad \downarrow(n+1) = \{n\}$$

for $n \in \mathbb{N}$. The associated well-founded recursion principle thus states that, given a number $r_0 \in \mathbb{N}$ and a function $r : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, there is a unique function $f : \mathbb{N} \to \mathbb{N}$ such that

$$f(0) = r_0 \qquad\qquad\qquad f(n+1) = r(n, f(n))$$

for every $n \in \mathbb{N}$. If we use the following generic notation for the image of n under the function associated to r_0 and r,

$$f(n) = \mathrm{rec}(n, r_0, r)$$

we have

$$\mathrm{rec}(0, r_0, r) = r_0 \qquad\qquad \mathrm{rec}(n+1, r_0, r) = r(n, \mathrm{rec}(n, r_0, r))$$

which are precisely the rules for the usual recursor associated to natural numbers in λ-calculus, see section 4.3.6.

The well-founded subterm order. We now explain that the kind of recursion which is supported in Agda is a particular case of the one given in theorem 6.8.6.6. Suppose given a first order signature Σ and consider the set $\mathcal{T}_\Sigma$ of terms it generates, see section 5.1.2. Given a term t, we write $|t|$ for its *size*, defined as the number of operators it contains:

$$|f(t_1, \ldots, t_n)| = 1 + \sum_i |t_i| \qquad\qquad |x| = 0$$

Given two terms s and t, we write $s < t$ when s is a strict subterm of t. Note that $s < t$ implies $|s| < |t|$.

Lemma 6.8.6.8. The relation $<$ on terms is well-founded.

Proof. Suppose that there is an infinite sequence of terms t_i such that

$$t_0 > t_1 > t_2 > \dots$$

the we have an infinite strictly decreasing sequence of natural numbers

$$|t_0| > |t_1| > |t_2| > \dots$$

which is impossible because $>$ is well-founded on $\mathbb{N}$, see example 6.8.6.1. $\square$

The recursion principle associated to this well-founded order is essentially the one used in Agda in order to define function: a function can be defined by recursion from the current value of the argument, as well as the image of strict subterms.

Accessible elements. Suppose given a set A and a relation R on it, not supposed to be well-founded. We can define a subset of A, written $\mathrm{Acc}_R(A)$, which is the largest subset of A on which well-founded induction and recursion works, as follows.

A subset $B \subseteq A$ is *R-closed* when, for every $x \in A$, $\downarrow x \subseteq B$ implies $x \in B$: if an element has all its predecessors in B then it is also in B. We define the set $\mathrm{Acc}_R(A)$ as the smallest R-closed subset of A (such a set exists since it can be obtained as the intersection of all R-closed subsets of A). An element of A is *accessible* with respect to R when it belongs to $\mathrm{Acc}_R(A)$.

Theorem 6.8.6.9. A relation R on a set A is well-founded if and only if every element of A is accessible, i.e. $A = \mathrm{Acc}_R(A)$.

In particular, given a relation R on a set A, the restriction of R to $\mathrm{Acc}_R(A)$ is always well-founded.

Example 6.8.6.10. In $\mathbb{N}$ equipped with the relation $\prec$ or $<$, every element is accessible.

Example 6.8.6.11. On the set $\mathbb{Z}$ equipped with the relation $<$, no element is accessible.

Example 6.8.6.12. On the set $\mathbb{R}$ equipped with the relation $\prec$ such that $x \prec x+1$ for every x in $\mathbb{R}$, the set of positive reals, the set $\mathrm{Acc}_\prec(\mathbb{R} \setminus \{-1\})$ is $\mathbb{N}$.

Well-foundedness in Agda. Although Agda only implements well-founded recursion on the subterm order natively, we will see that this is enough for most applications: we can encode general well-founded recursion (at least for most usual well-founded orders, some will always stay out of reach by arguments similar to those in section 6.8.3).

Recall from section 6.5.9 that we can define a type `Rel A` of relations on a type `A`, which is `A → A → Set`. For instance, the strict order on natural numbers is a relation:

```
_<_ : Rel ℕ
m < n = suc m ≤ n
```

In order to define the predicate of being well-founded for a relation, the direct definition of well-foundedness is difficult to implement in Agda, inelegant and difficult to use (mainly because it is defined using a negation). A much more satisfactory approach consists in taking the characterization given in theorem 6.8.6.9 as a definition.

Following the module `Induction.WellFounded` of the standard library, we define the predicate of being accessible as

```
data Acc {A : Set} (_<_ : Rel A) (x : A) : Set where
  acc : ((y : A) → y < x → Acc _<_ y) → Acc _<_ x
```

Given a relation `R` on a type `A` and an element `x` of type `A`, having a proof of `Acc R x` means that `x` is accessible with respect to `R`. Namely, the inductive definition states that the predicate `Acc R` is defined as the smallest one such that `Acc R x` whenever we have `Acc R y` for every predecessor `y` of `x`, which is precisely the definition of accessibility. Finally, we define a relation to be well-founded when every element is accessible with respect to it:

```
WellFounded : {A : Set} → (_<_ : Rel A) → Set
WellFounded {A} _<_ = (x : A) → Acc _<_ x
```

Natural numbers are well-founded. As an instance of the above formalization, let us show that the strict order `<` on natural numbers is well-founded. It turns out that the usual definition of the order (see section 6.5.9) is not very well-suited for the induction we want to perform, and it proves simpler to use the following alternative definition of the order:

```
data _≤'_ (m : ℕ) : ℕ → Set where
  ≤'-refl :                   m ≤' m
  ≤'-step : {n : ℕ} → m ≤' n → m ≤' suc n
```

the associated strict order being defined by

```
_<'_ : ℕ → ℕ → Set
m <' n = suc m ≤' n
```

We can then show that the relation `<'` is well-founded as follows:

```
<'-wellFounded : WellFounded _<'_
<'-wellFounded n = acc (lem n)
  where
  lem : (n m : ℕ) → m <' n → Acc _<'_ m
  lem (suc n) _ ≤'-refl        = <'-wellFounded n
  lem (suc n) m (≤'-step m<'n) = lem n m m<'n
```

Of course, one is usually rather interested in the fact that the usual definition `<` of the strict order is well-founded. This can either be deduced from the fact that the relations `<` and `<'` are equivalent, or it can be shown directly. Namely, the following lemma can be shown on the usual definition of the partial order

```
≤-last : {m n : ℕ} → m ≤ n → m ≡ n ⊎ m < n
≤-last {n = zero}  z≤n = inj₁ refl
≤-last {n = suc n} z≤n = inj₂ (s≤s z≤n)
```

```
≤-last (s≤s m≤n) with ≤-last m≤n
≤-last (s≤s m≤n) | inj₁ m≡n = inj₁ (cong suc m≡n)
≤-last (s≤s m≤n) | inj₂ m<n = inj₂ (s≤s m<n)
```

from which the above proof can be adapted to show that < is well-founded:

```
<-wellFounded : WellFounded _<_
<-wellFounded n = acc (lem n)
  where
  lem : (n m : ℕ) → m < n → Acc _<_ m
  lem (suc n) m  m<n with ≤-last m<n
  lem (suc n) _ _ | inj₁ refl      = <-wellFounded n
  lem (suc n) m _ | inj₂ (s≤s m<n) = lem n m m<n
```

Well-founded definition of bits. As an application, we shall define our favorite bits functions by well-founded recursion on the order < on natural numbers. Given an argument n, in order to perform recursive calls on smaller arguments (with respect to the < order), we add an argument of type Acc _<_ n to the naive definition of bits, which is a proof that n is accessible, i.e. that all the elements strictly smaller than n are also accessible. Namely, an element of this type is of the form acc a with a of type

$$(m : ℕ) → m < n → Acc \_<\_ m$$

It is used here as a witness that the function is terminating. For instance, the function bits becomes

```
bits-wf : (n : ℕ) → Acc _<_ n → ℕ
bits-wf zero     _       = zero
bits-wf (suc n) (acc a) =
  suc
    (bits-wf
      (div2 (suc n))
      (a (div2 (suc n)) (s≤s (≤-div2-suc n))))
```

In order to perform the recursive call on div2 (suc n), we have to show that this number is accessible, which is deduced from the fact that div2 (suc n) < suc n holds, as explained above. Finally, the usual bits function, without the extra argument, is defined by providing the proof that every natural number is accessible as second argument (which is precisely the fact that the relation < is well-founded on natural numbers):

```
bits : ℕ → ℕ
bits n = bits-wf n (<-wellFounded n)
```

Well-founded recursion without accessibility. In practice, it is quite annoying to require that the "average Agda user" should understand the definition of the accessibility predicate, so that the standard library defines the following function in the module Data.Nat.Induction, which expresses well-founded recursion in a way which does not require using arguments of type Acc. It is also nicer to read, since it precisely corresponds to the strong induction principle, as formulated in example 6.8.6.5:

```
<-rec : (P : ℕ → Set) →
        ((n : ℕ) → ((m : ℕ) → m < n → P m) → P n) →
        (n : ℕ) → P n
<-rec P r n = lem n (<-wellFounded n)
  where
  lem : (n : ℕ) → Acc _<_ n → P n
  lem n (acc a) = r n (λ m m<n → lem m (a m m<n))
```

In the end, this is all you will ever need to define functions by well-founded recursion on natural numbers in practice (and of course the same can be performed for any well-founded relation).

For instance, the function computing the number of bits of a natural number n can be implemented by first adding to the naive implementation a new argument, which is a proof that the function is already defined for strictly smaller arguments than the current natural number: this argument will have type

```
(m : ℕ) → m < n → ℕ
```

and provides a function to compute the recursive calls on strictly smaller arguments. We thus obtain the following function:

```
bits-rec : (n : ℕ) → ((m : ℕ) → m < n → ℕ) → ℕ
bits-rec zero    r = zero
bits-rec (suc n) r = suc (r (div2 (suc n)) (s≤s (≤-div2-suc n)))
```

Finally, we can deduce an implementation of the expected bits function by using it as an argument of `<-rec`:

```
bits : ℕ → ℕ
bits = <-rec (λ n → ℕ) bits-rec
```

6.8.7 Division and modulo. As another classic illustration of the above techniques, we shall implement euclidean division. It associates to each pair of natural numbers m and n, with $n > 0$, a pair of natural numbers q and r, respectively called the *quotient* and *remainder* of the division of m by n such that

$$m = q \times n + r \qquad \text{and} \qquad r < n \qquad (6.2)$$

Numbers satisfying these properties can be shown to be unique, so that this is a proper specification for euclidean division. Their traditional notations are respectively

$$q = m/n \qquad\qquad r = m \bmod n$$

and they can be computed using the following classic algorithm, here implemented in OCaml:

```
let rec euclid m n =
  if m < n then (0, m) else
    let (q, r) = euclid (m - n) n in
    (q + 1, r)
```

Well-founded definition: external approach. The direct translation of the above algorithm is naturally performed by well-founded induction on m, and is justified by the fact that the recursive call is performed on m - n which is strictly smaller than m because we assume that n is strictly positive. The type of the division operation we want to define is

$$\text{(m n : ℕ) → 0 < n → ℕ × ℕ}$$

taking m and n as arguments, as well as a proof of 0 < n, and returning the pair (q , r) as result. Since we perform the induction on the first argument, we are going to define, by well-founded recursion on m, a function of type

$$\text{(n : ℕ) → 0 < n → ℕ × ℕ}$$

for every natural number m. In order not to have to type this every time, we define the notation `Euclid m` for this type:

```
Euclid : ℕ → Set
Euclid m = (n : ℕ) → 0 < n → ℕ × ℕ
```

We can then implement euclidean division by well-founded recursion, following the above definition: when computing the result of the division of m and n, we first check whether $m < n$ holds or not, and provide an answer appropriately, which requires performing a recursive call in the case $m \not< n$ (which requires additional code because we now have to provide a proof that $m - n < n$ holds). The definition is

```
div : (m : ℕ) → Euclid m
div m = <-rec Euclid rec m
  where
  rec : (m : ℕ) → ((m' : ℕ) → m' < m → Euclid m') → Euclid m
  rec m f n 0<n with m <? n
  rec m f n 0<n | yes m<n = zero , m
  rec m f n 0<n | no  m≮n with
    f (m ∸ n) (m∸n<m m n (<-trans¹ 0<n (≮⇒≥ m≮n)) 0<n) n 0<n
  rec m f n 0<n | no  m≮n | q , r = suc q , r
```

and uses the following auxiliary lemma (in addition to those already present in the standard library):

```
m∸n<m : (m n : ℕ) → 0 < m → 0 < n → m ∸ n < m
m∸n<m (suc m) (suc n) _ _ = s≤s (m∸n≤m m n)
```

Finally, it can be shown that this implementation is correct, in the sense that it satisfies the specification (6.2). Formally, we can show

```
div-correct :
  (m n : ℕ) (0<n : 0 < n) →
  m ≡ proj₁ (div m n 0<n) * n + (proj₂ (div m n 0<n)) ×
  (proj₂ (div m n 0<n)) < n
```

However, the proof is not obvious, due to the use of the well-founded induction and a more satisfactory approach is detailed in next section.

Well-founded definition: intrinsic approach. We now present the intrinsic approach which, as explained in section 6.7.1, consists in enriching the type, so that the implementation is correct by definition (as opposed to being proved correct after being defined). We first define a type corresponding to the specification of euclidean division:

```
Euclid : ℕ → Set
Euclid m = (n : ℕ) → 0 < n →
           Σ ℕ (λ q → Σ ℕ (λ r → m ≡ q * n + r × r < n))
```

so that euclidean division will have type

$$(m : ℕ) → \text{Euclid } m$$

and thus consist of a function which takes as arguments

- a natural number `m`,

- a natural number `n`,

- a proof of `0 < n`,

and will return a dependent 4-uple consisting of

- a natural number `q` (the quotient),

- a natural number `r` (the remainder),

- a proof of `m ≡ q * n + r`,

- a proof of `r < n`,

which is a type theoretic description of the specification (6.2). The implementation is very similar to the above one, except that we now have to return, in addition to the quotient and the remainder, the two proofs indicated above which show that those results are correct. The full code is

```
div : (m : ℕ) → Euclid m
div m = <-rec Euclid rec m
  where
  rec : (m : ℕ) → ((m' : ℕ) → m' < m → Euclid m') → Euclid m
  rec m f n 0<n with m <? n
  rec m f n 0<n | yes m<n = zero , m , refl , m<n
  rec m f n 0<n | no  m≮n with
    f (m ∸ n) (m∸n<m m n (<-trans¹ 0<n (≮⇒≥ m≮n)) 0<n) n 0<n
  rec m f n 0<n | no  m≮n | q , r , e , r<n =
    suc q , r , lem , r<n
    where
    lem : m ≡ n + q * n + r
    lem = begin
      m                ≡⟨ sym (m+[n∸m]≡n (≮⇒≥ m≮n)) ⟩
      n + (m ∸ n)      ≡⟨ cong (λ x → n + x) e ⟩
      n + (q * n + r) ≡⟨ sym (+-assoc n (q * n) r) ⟩
      n + q * n + r    ∎
```

Instead of trying to read it, the reader is urged to try this by himself instead.

Inductive definition. For general culture, we shall also mention that it is also possible to implement euclidean division by structural definition, avoiding the use of well-founded induction. This is the approach followed in Agda's standard library, in `Data.Nat.DivMod`. The trick consists in adding two extra arguments `q` and `r'` to the naive function, which will keep track of the quotient and remainder (or, more precisely, `n` minus the remainder). Namely, given `m` and `n`, we will perform our definition by induction on `m`. Initially `q` is 0 and `r'` is `n`, each time `m` is decreased by one,

- if `r'` is strictly positive, we decrease it by one,

- if `r'` is 0, we increase `q` by one and reset `r'` to `n`.

Formally, the code follows:

```
euclid : (m n q r' : ℕ) → ℕ × ℕ
euclid zero     n q r'        = q , n ∸ r'
euclid (suc m) n q zero     = euclid m n (suc q) n
euclid (suc m) n q (suc r') = euclid m n q r'
```

It can be shown that, for every `m` and `n`, the result of

```
euclid m n zero n
```

computes the quotient and remainder of `m` by `suc n` (we consider here `suc n` in order to ensure that the denominator is non-zero).

Exercise 6.8.7.1. Show that this function is correct.

Exercise 6.8.7.2. Give an intrinsic inductive definition of euclidean division.

Formalization of important results

In this chapter, we sketch the formalization in Agda of important concepts and results presented in this book: type safety (section 7.1), natural deduction (section 7.2), λ-calculus (section 7.3), combinatory logic (section 7.4), simply-typed λ-calculus (section 7.5).

7.1 Safety of a simple language

In section 1.4.3, we have studied a simple typed language consisting of expressions manipulating booleans and integers and have shown the two fundamental properties satisfied by this language: subject reduction and progress. We now explain how those can be formalized in Agda. We begin by importing the required libraries, and renaming and hiding symbols so that we can redefine those used by the standard library:

```
open import Data.Bool hiding   (if_then_else_ ; _≟_ ; _<_ ; _<?_)
open import Data.Nat  renaming (_+_ to _+ℕ_ ; _<?_ to _<?ℕ_)
                      hiding   (_<_)
open import Relation.Nullary
```

The language. A value in the language is either a natural number or a boolean:

```
data Value : Set where
  VNat  : ℕ    → Value
  VBool : Bool → Value
```

A program is either a value, an addition, a comparison, or a conditional branching:

```
data Prog : Set where
  V              : Value → Prog
  _+_            : Prog → Prog → Prog
  _<_            : Prog → Prog → Prog
  if_then_else_ : Prog → Prog → Prog → Prog
```

and we assign priorities to these constructors, in order to ease the writing of programs:

```
infix 50 _+_
infix 40 _<_
infix 30 if_then_else_
```

We will need to compare natural numbers with this function, which returns a boolean depending on whether the first is strictly smaller than the second or not:

```
_<?_ : ℕ → ℕ → Bool
m <? n with m <?ℕ n
(m <? n) | yes _ = true
(m <? n) | no  _ = false
```

We then define the reduction relation as an inductive binary predicate _⇒_, so that, given programs p and q, a proof of p ⇒ q corresponds to a derivation of ⊢ p ⟶ q using the rules of figure 1.1: we add one constructor to this inductive predicate for each inference rule.

```
data _⇒_ : Prog → Prog → Set where
  ⇒-Add   : (m n : ℕ) →
            V (VNat m) + V (VNat n) ⇒ V (VNat (m +ℕ n))
  ⇒-Add-l : {p p' : Prog} → p ⇒ p' → (q : Prog) →
            p + q ⇒ p' + q
  ⇒-Add-r : {q q' : Prog} → (p : Prog) → q ⇒ q' →
            p + q ⇒ p + q'
  ⇒-Lt    : (m n : ℕ) →
            V (VNat m) < V (VNat n) ⇒ V (VBool (m <? n))
  ⇒-Lt-l  : {p p' : Prog} → p ⇒ p' → (q : Prog) →
            p < q ⇒ p' < q
  ⇒-Lt-r  : {q q' : Prog} → (p : Prog) → q ⇒ q' →
            p < q ⇒ p < q'
  ⇒-If    : {p p' : Prog} → p ⇒ p' → (q r : Prog) →
            if p then q else r ⇒ if p' then q else r
  ⇒-If-t  : (p q : Prog) →
            if V (VBool true) then p else q ⇒ p
  ⇒-If-f  : (p q : Prog) →
            if V (VBool false) then p else q ⇒ q
```

Typing. We now define the typing system of our language, starting with the definition of a type which is either a natural number or a boolean:

```
data Type : Set where
  TNat TBool : Type
```

We then define the typing relation as an inductive binary predicate ⊢_::_, so that a proof of ⊢ p :: A for a program p and type A corresponds precisely to a proof of ⊢ p : A using the type inference rules given in section 1.4.3:

```
data ⊢_::_ : Prog → Type → Set where
  ⊢-Nat  : (n : ℕ) →
           ⊢ V (VNat n) :: TNat
  ⊢-Bool : (b : Bool) →
           ⊢ V (VBool b) :: TBool
  ⊢-Add  : {p q : Prog} → ⊢ p :: TNat → ⊢ q :: TNat →
           ⊢ p + q :: TNat
  ⊢-Lt   : {p q : Prog} → ⊢ p :: TNat → ⊢ q :: TNat →
           ⊢ p < q :: TBool
  ⊢-If   : {p q r : Prog} {A : Type} →
           ⊢ p :: TBool → ⊢ q :: A → ⊢ r :: A →
           ⊢ if p then q else r :: A
```

Type uniqueness. This formalization of typing as an inductive predicate has a very interesting byproduct: the dependent pattern matching algorithm knows, given the constructor of a program, the possible types this program can have (and conversely, given a type, the possible program constructors which will give rise to this type). Thanks to this, showing type uniqueness (theorem 1.4.3.1) is simple:

```
tuniq : {p : Prog} {A A' : Type} → ⊢ p :: A → ⊢ p :: A' → A ≡ A'

tuniq (⊢-Nat n)    (⊢-Nat .n)     = refl
tuniq (⊢-Bool b)   (⊢-Bool .b)    = refl
tuniq (⊢-Add t u)  (⊢-Add t' u')  = refl
tuniq (⊢-Lt t u)   (⊢-Lt t' u')   = refl
tuniq (⊢-If t u v) (⊢-If t' u' v') = tuniq v v'
```

For instance, in the first case (the program is a natural number), Agda infers that its type is necessarily `TNat` and therefore `A` and `A'` must be equal (to `TNat`).

Subject reduction. The subject reduction theorem (theorem 1.4.3.2) states that if a program p reduces to p' and p admits the type A, then p' also admits the type A. The proof is most easily done by induction on the derivation of $p \longrightarrow p'$:

```
sred : {p p' : Prog} {A : Type} → (p ⇒ p') → ⊢ p :: A → ⊢ p' :: A

sred (⇒-Add m n)    (⊢-Add _ _)    = ⊢-Nat (m +ℕ n)
sred (⇒-Add-l r q)  (⊢-Add t t')   = ⊢-Add (sred r t) t'
sred (⇒-Add-r p r)  (⊢-Add t t')   = ⊢-Add t (sred r t')
sred (⇒-Lt m n)     (⊢-Lt t t')    = ⊢-Bool (m <? n)
sred (⇒-Lt-l r q)   (⊢-Lt t t')    = ⊢-Lt (sred r t) t'
sred (⇒-Lt-r p r)   (⊢-Lt t t')    = ⊢-Lt t (sred r t')
sred (⇒-If p q r)   (⊢-If t t₁ t₂) = ⊢-If (sred p t) t₁ t₂
sred (⇒-If-t p q)   (⊢-If t t₁ t₂) = t₁
sred (⇒-If-f p q)   (⊢-If t t₁ t₂) = t₂
```

Progress. The last important property of our typed language is progress (theorem 1.4.3.3) which states that a typable program is either a value or reduces to some other program. Given a program p which admits a type A, the proof is performed on the derivation of $\vdash p : A$:

```
prgs : {p : Prog} {A : Type} → ⊢ p :: A →
       Σ Value (λ v → p ≡ V v) ⊎ Σ Prog (λ p' → p ⇒ p')

prgs (⊢-Nat n)  = inj₁ (VNat n , refl)
prgs (⊢-Bool b) = inj₁ (VBool b , refl)
prgs (⊢-Add t t') with prgs t
prgs (⊢-Add t t') | inj₁ (v , e) with prgs t'
prgs (⊢-Add t t') | inj₁ (VNat m , refl) | inj₁ (VNat n , refl) =
  inj₂ (V (VNat (m +ℕ n)) , ⇒-Add m n)
prgs (⊢-Add t ()) | inj₁ (VNat m , refl) | inj₁ (VBool b , refl)
prgs (⊢-Add () t') | inj₁ (VBool b , refl) | inj₁ (v' , refl)
prgs (⊢-Add {p} {q} t t') | inj₁ (v , e) | inj₂ (q' , r) =
```

```
  inj₂ (p + q' , ⇒-Add-r p r)
prgs (⊢-Add {p} {q} t t') | inj₂ (p' , r) =
  inj₂ ((p' + q) , ⇒-Add-l r q)
prgs (⊢-Lt t t') with prgs t
prgs (⊢-Lt t t') | inj₁ (VNat m , refl) with prgs t'
prgs (⊢-Lt t t') | inj₁ (VNat m , refl) | inj₁ (VNat n , refl) =
  inj₂ ((V (VBool (m <? n))) , ⇒-Lt m n)
prgs (⊢-Lt t ()) | inj₁ (VNat m , refl) | inj₁ (VBool b , refl)
prgs (⊢-Lt {p} {q} t t') | inj₁ (VNat m , refl) | inj₂ (q' , r) =
  inj₂ (V (VNat m) < q' , ⇒-Lt-r (V (VNat m)) r)
prgs (⊢-Lt () t') | inj₁ (VBool b , refl)
prgs (⊢-Lt {p} {q} t t') | inj₂ (p' , r) =
  inj₂ (p' < q , ⇒-Lt-l r q)
prgs (⊢-If t t₁ t₂) with prgs t
prgs (⊢-If () t₁ t₂) | inj₁ (VNat x , refl)
prgs (⊢-If {_} {q} {r} t t₁ t₂) | inj₁ (VBool false , refl) =
  inj₂ (r , ⇒-If-f q r)
prgs (⊢-If {_} {q} {r} t t₁ t₂) | inj₁ (VBool true , refl) =
  inj₂ (q , ⇒-If-t q r)
prgs (⊢-If {p} {q} {r} t t₁ t₂) | inj₂ (p' , pr) =
  inj₂ (if p' then q else r , ⇒-If pr q r)
```

Exercise 7.1.0.1. Formalize type inference and show that

- it is correct: if a type is inferred for a program then the program actually admits this type,

- it is complete: if a program is typable then type inference will return a type.

7.2 Natural deduction

The proofs in natural deduction are presented in section 2.2, we now briefly present how those can be formalized in Agda. For conciseness, we only present here the implicative fragment.

Formulas. A formula is either a variable (whose name is given by a natural number) or the implication of two formulas (see section 2.2.1):

```
data Formula : Set where
  X   : ℕ → Formula
  _⇒_ : Formula → Formula → Formula
```

Contexts. Next, we formalize a context as being either the empty context ε or a pair Γ , A consisting of a context Γ and a formula A:

```
data Context : Set where
  ε : Context
  _,_ : (Γ : Context) → (A : Formula) → Context
```

We could also have formalized contexts as lists of formulas, but the above formalization allows for a slightly more natural notation. We write Γ ,, Δ for the concatenation of two contexts Γ and Δ:

```
_,,_ : Context → Context → Context
Γ ,, ε = Γ
Γ ,, (Δ , A) = (Γ ,, Δ) , A
```

Provable sequents. We can define the type Γ ⊢ A of provable sequents as an inductive predicate, with one constructor corresponding to each inference rule:

```
data _⊢_ : Context → Formula → Set where
  ax : ∀ {Γ A Γ'} → Γ , A ,, Γ' ⊢ A
  ⇒E : ∀ {Γ A B}  → Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
  ⇒I : ∀ {Γ A B}  → Γ , A ⊢ B → Γ ⊢ A ⇒ B
```

(the axiom rule and the elimination and introduction rules for implication). This formalization is not very convenient because the argument of the constructor `ax` uses concatenation ",," which is a function and not a type constructor, and will prevent pattern matching from working: unlike a constructor, this function does not have the property that

$$\Gamma \ ,, \ \Delta = \Gamma' \ ,, \ \Delta' \qquad \text{implies} \qquad \Gamma = \Gamma' \quad \text{and} \quad \Delta = \Delta'$$

In order to overcome this problem, we chose instead to formalize provable sequents as

```
data _⊢_ : Context → Formula → Set where
  ax : ∀ {Γ A} → Γ , A ⊢ A
  wk : ∀ {Γ A B} → Γ ⊢ B → Γ , A ⊢ B
  ⇒E : ∀ {Γ A B} → Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
  ⇒I : ∀ {Γ A B} → Γ , A ⊢ B → Γ ⊢ A ⇒ B
```

which consists in replacing the usual axiom rule

$$\frac{}{\Gamma, A, \Gamma' \vdash A} \ (\text{ax})$$

by the two rules

$$\frac{}{\Gamma, A \vdash A} \ (\text{ax}) \qquad\qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B} \ (\text{wk})$$

which give rise to an equivalent logical system.

Admissible rules. This can be used to show that the usual rules are admissible. For instance, we can prove that the contraction rule

$$\frac{\Gamma, A, A, \Gamma' \vdash B}{\Gamma, A, \Gamma' \vdash B} \ (\text{contr})$$

is admissible (see section 2.2.7) by induction, both on the context Γ' and on the proof of the premise, by

```
cont : ∀ {Γ A B} → ∀ Γ' → Γ , A , A ,, Γ' ⊢ B → Γ , A ,, Γ' ⊢ B
cont ε          ax       = ax
cont ε          (wk p)   = p
cont ε          (⇒E p q) = ⇒E (cont ε p) (cont ε q)
cont ε          (⇒I p)   = ⇒I (cont (ε , _) p)
cont (Γ' , A) ax         = ax
cont (Γ' , A) (wk p)     = wk (cont Γ' p)
cont (Γ' , A) (⇒E p q) = ⇒E (cont (Γ' , A) p) (cont (Γ' , A) q)
cont (Γ' , A) (⇒I p)     = ⇒I (cont (Γ' , A , _) p)
```

Similarly, admissibility of the cut rule

$$\frac{\Gamma \vdash A \qquad \Gamma, A, \Gamma' \vdash B}{\Gamma, \Gamma' \vdash B}$$

is shown by

```
cut : ∀ {Γ A B} → ∀ Γ' → Γ ⊢ A → Γ , A ,, Γ' ⊢ B → Γ ,, Γ' ⊢ B
cut ε          p ax       = p
cut ε          p (wk q)   = q
cut ε          p (⇒E q r) = ⇒E (cut ε p q) (cut ε p r)
cut ε          p (⇒I q)   = ⇒I (cut (ε , _) p q)
cut (Γ' , A) p ax         = ax
cut (Γ' , A) p (wk q)     = wk (cut Γ' p q)
cut (Γ' , A) p (⇒E q r) = ⇒E (cut (Γ' , A) p q) (cut (Γ' , A) p r)
cut (Γ' , A) p (⇒I q)     = ⇒I (cut (Γ' , A , _) p q)
```

Exercise 7.2.0.1. Formalize the admissibility of the other rules presented in section 2.2.7.

7.3 Pure λ-calculus

In this section, we present a formalization of λ-calculus in Agda, using de Bruijn indices.

7.3.1 Naive approach. We can first think of directly translating the definition of λ-terms given in section 3.1. We suppose fixed an infinite set of variables (say, the strings),

```
Var : Set
Var = String
```

and define the syntax of λ-terms as

```
data Tm : Set where
  var   : Var → Tm
  _·_   : Tm  → Tm → Tm
  λ_,_  : Var → Tm → Tm
```

meaning that a term is either of the form var x (the variable x), or t · u (the application of t to u) or λ x , t (the function which to x associates t). The

weird choice of symbols in the last case comes from the fact that the dot (.) and lambda (λ) are reserved in Agda.

We could proceed in this way, but one should remember that λ-terms are not terms generated by the above syntax, but rather of the quotient under α-equivalence (section 3.1.3). This means that we will have to define this equivalence and show that all the constructions we are going to make are compatible with it. This is rather long and painful.

Exercise 7.3.1.1. Try to properly define β-reduction with this formalization.

7.3.2 De Bruijn indices. In order to efficiently handle the α-conversion problem, we are going to use de Bruijn indices for variables, as presented in section 3.6.2. We thus define terms as

```
data Tm : Set where
  var : ℕ → Tm
  _·_ : Tm → Tm → Tm
  ƛ_  : Tm → Tm
```

A term can thus be in one of the following forms:

– `var x`: the x-th variable with x a natural number,

– `t · u`: the application of a term t to a term u,

– `ƛ t`: the abstraction of the 0-th variable in t.

Lifting. The next thing we want to do is define β-reduction, but before being able to do this, we first need to introduce helper functions in order to explicitly manipulate variables, following section 3.6.2.

The first one is *lifting* which can be thought of as creating a fresh variable numbered x. After performing this operation, all the variable indices y which are greater than x have to be increased by one in order to make room for x. The new index of y after the creation of x is written $\uparrow_x y$ and defined by

$$\uparrow_x y = \begin{cases} y & \text{if } y < x, \\ y+1 & \text{if } y \geqslant x. \end{cases}$$

In Agda, this function can be defined by

```
↑ : ℕ → ℕ → ℕ
↑ zero    y       = suc y
↑ (suc x) zero    = zero
↑ (suc x) (suc y) = suc (↑ x y)
```

and we write `↑ x y` for $\uparrow_x y$.

Conversely, the *unlifting* operation consists in removing an unused variable x. After the removal, all the variable indices y which are greater than x have to be decreased by one in order to fill in the "empty space" leaved by x. Their new index will thus be $\downarrow_x y$, defined by

$$\downarrow_x y = \begin{cases} y & \text{if } y < x, \\ y-1 & \text{if } y > x. \end{cases}$$

The function is not defined when $y = x$, because we have supposed that the variable x is not used. In Agda, this can be defined as

```
↓ : (x y : ℕ) → x ≢ y → ℕ
↓ zero    zero    ¬p = ⊥-elim (¬p refl)
↓ zero    (suc y) ¬p = y
↓ (suc x) zero    ¬p = zero
↓ (suc x) (suc y) ¬p = suc (↓ x y (λ p → ¬p (cong suc p)))
```

and we write ↓ x y p for $\downarrow_x y$: in addition to x and y, the Agda function takes a third argument p which is a proof that x is different from y.

The above lifting operation can be extended to λ-terms. Given a variable x and a λ-term t, the term $\uparrow_x t$ obtained after creating a fresh variable x will be written here wk x t, because it is thought of as some form for *weakening* for the term t. The weakening function wk is defined here by

```
wk : ℕ → Tm → Tm
wk x (var y)  = var (↑ x y)
wk x (t · t') = wk x t · wk x t'
wk x (ƛ t)    = ƛ (wk (suc x) t)
```

This definition uses lifting on variables, recursively applies weakening for applications and abstractions. There is a subtlety for the last case: since the abstraction binds the variable 0 in a term t, a variable x in $\lambda.t$ corresponds to a the variable $x + 1$ in t, which explains why we have to increase by one the weakened variable when going under abstractions.

Substitution. We can then define *substitution*, as detailed in section 3.6.2:

```
_[_/_] : Tm → Tm → ℕ → Tm
var y    [ u / x ] with x ≟ y
(var y   [ u / _ ]) | yes _ = u
(var y   [ u / x ]) | no ¬p = var (↓ x y ¬p)
(t · t') [ u / x ]  = (t [ u / x ]) · (t' [ u / x ])
(ƛ t)    [ u / x ]  = ƛ (t [ wk 0 u / ↑ 0 x ])
```

The two subtle corner cases when substituting a variable x by a term u in a term t are:

- all the variables different from x have to be renumbered using ↓ since substitution removes all occurrences of x, which is supposed not to be free in u,

- when going under an abstraction, the term u has to be weakened using wk and the variable x has to be renamed using ↑ in order to account for the fact that the variable 0 is bound by the abstraction.

The β-reduction. Once substitution defined, we can define *β-reduction* by following the usual definition, which is given in section 3.2.1: we implement it as an inductive predicate with one constructor for each inference rule defining the reduction.

```
data _↝_ : Tm → Tm → Set where
  ↝β : {t u   : Tm}         → (λ t) · u ↝ t [ u / 0 ]
  ↝l : {t t' u : Tm} → t ↝ t' →     t · u ↝ t' · u
  ↝r : {t u u' : Tm} → u ↝ u' →     t · u ↝ t  · u'
  ↝λ : {t t'   : Tm} → t ↝ t' →      λ t ↝ λ t'
```

The *iterated β-reduction* relation ↝* is the reflexive and transitive closure of the β-reduction relation ↝. In order to define it, we can use the module

`Relation.Binary.Construct.Closure.ReflexiveTransitive`

of the standard library which defines the closure of any relation by

```
data Star {A : Set} (R : Rel A) : Rel A where
  ε   : {x : A} → Star R x x
  _◂_ : {x y z : A} → R x y → Star R y z → Star R x z
```

which is based on the characterization given in lemma A.1.2.1. We can therefore define the relation ↝* by

```
_↝*_ : Tm → Tm → Set
_↝*_ = Star _↝_
```

Church natural numbers. As explained in section 3.3.4, we can encode a natural number n as the term $\lambda f.x f^n x$. We can define a function nat' which, given a natural number n and two variables f and x, produces the term $f^n x$ by induction on n by

```
nat' : (n : ℕ) (f x : ℕ) → Tm
nat' 0 f x = var x
nat' (suc n) f x = var f · nat' n f x
```

and the Church encoding of natural numbers can then be defined as

```
nat : ℕ → Tm
nat n = λ (λ (nat' n 1 0))
```

The term computing the successor of a natural number can then be defined as

```
succ : Tm
succ = λ λ λ (var 1 · (var 2 · var 1 · var 0))
```

and the one computing the addition of two natural numbers as

```
add : Tm
add = λ λ λ λ (var 3 · var 1 · (var 2 · var 1 · var 0))
```

Exercise 7.3.2.1. Show that those two last terms are correct, in the sense that they actually compute the successor and addition of natural numbers, i.e. we have

```
succing : (n : ℕ) → succ · nat n ↝* nat (suc n)
```

and

```
adding : (m n : ℕ) → add · nat m · nat n ↝* nat (m + n)
```

In order to do so, you should be prepared to prove quite a few lemmas about substitution and lifting (see below).

7.3.3 Keeping track of free variables. As a side note, let us present a refinement of the above formalization. Since the implementation of λ-calculus with de Bruijn indices is quite technical and error-prone, it is sometimes useful to have the most precise type possible, in order to detect errors early. One way to do so is to keep track of the free variables used in a term. Instead of defining the type `Tm` of all terms, we can define, for each natural number `n`, the type `Tm n` of terms whose free variables `x` are natural numbers such that $0 \leqslant x < n$. This last constraint is conveniently described by requiring that `x` is an element of type `Fin n`, see section 6.4.8. This refinement of the formalization avoids inadvertently getting the wrong names for free variables and allows for reasoning by induction on the number of free variables in terms. We thus define terms as

```
data Tm (n : ℕ) : Set where
  var : Fin n → Tm n
  _·_ : Tm n → Tm n → Tm n
  λ_  : Tm (suc n) → Tm n
```

In the last case, the term `t` should have at least one free variable, so that its type is of the form `Fin (suc n)`, and will have one less free variable since one variable was bound, so that the return type is `Fin n`.

Most previous functions can be adapted directly to this setting, so that we only give the refined types for those. The type now makes it clear that lifting inserts a fresh variable

```
↑ : {n : ℕ} → Fin (suc n) → Fin n → Fin (suc n)
```

as well as does weakening

```
wk : {n : ℕ} → Fin (suc n) → Tm n → Tm (suc n)
```

and that unlifting removes a variables

```
↓ : {n : ℕ} (x y : Fin (suc n)) → x ≢ y → Fin n
```

as well as does substitution

```
_[_/_] : {n : ℕ} → Tm (suc n) → Tm n → Fin (suc n) → Tm n
```

Finally, the type of reduction should indicate that it preserves free variables:

```
_⟶_ {n : ℕ} : Tm n → Tm n → Set
```

The rest of the developments can be performed in this way. We do not present those here because they are more cumbersome to perform: in all the proofs, we have to show that the number of free variables is correctly handled. The formalization of section 7.5 is also quite close to this one: there, in addition to keeping track of the number of variables, we will also keep track of their type.

7.3.4 Normalization by evaluation. As another side note, the reader having read section 3.5.2 might think that it could be a good idea to use normalization by evaluation in order to implement β-reduction instead of de Bruijn indices. This suggests defining the following notions of value and neutral term:

```
  data Value : Set
  data Neutral : Set
  data Value where
    λ_  : (Value → Value) → Value
    N   : Neutral → Value
  data Neutral where
    var : Var → Neutral
    _·_ : Neutral → Value → Neutral
```

However, this definition is not accepted by Agda, which raises the following error:

```
Value is not strictly positive, because it occurs to the left of
an arrow in the type of the constructor λ_ in the definition of
Value.
```

This is explained in section 8.4.4, where we show that removing the associated restriction leads to Agda being inconsistent. We will see in section 7.5.3 that we can nevertheless implement normalization by evaluation for simply typed λ-calculus.

7.3.5 Confluence. Based on previous definitions, we now formalize one of the main results: the confluence of β-reduction, following the proof given in section 3.4 (see [Hue94] for an admirable other way to prove this).

The parallel β-reduction. We first define the parallel β-reduction by

```
data _⇉_ : Tm → Tm → Set where
  ⇉v : (x : ℕ)              →                   var x ⇉ var x
  ⇉β : {t t' u u' : Tm} → t ⇉ t' → u ⇉ u' → λ t · u ⇉ t' [ u' / 0 ]
  ⇉a : {t t' u u' : Tm} → t ⇉ t' → u ⇉ u' →   t · u ⇉ t' · u'
  ⇉λ : {t t'      : Tm} → t ⇉ t'            →     λ t ⇉ λ t'
```

which mirrors the definition of section 3.4.2.

Local confluence of the parallel β-reduction. The *local confluence* of parallel β-reduction (also called *diamond property*) states that given terms t, u and v such that t parallel β-reduces to both u and v, there exists a term w such that both u and v parallel β-reduce to w. The proof can be formalized in Agda by case analysis on the reductions of t to u and t to v, closely following the proof presented in lemma 3.4.3.5:

```
⇉-lc : {t u v : Tm} → t ⇉ u → t ⇉ v →
       Σ Tm (λ w → u ⇉ w × v ⇉ w)

⇉-lc (⇉v x)     (⇉v .x)   = var x , ⇉v x , ⇉v x
⇉-lc (⇉β r₁ r₂) (⇉β s₁ s₂) with ⇉-lc r₁ s₁ | ⇉-lc r₂ s₂
⇉-lc (⇉β r₁ r₂) (⇉β s₁ s₂) | w₁ , r₁' , s₁' | w₂ , r₂' , s₂' =
  w₁ [ w₂ / 0 ] , ⇉-sub 0 r₁' r₂' , ⇉-sub 0 s₁' s₂'
⇉-lc (⇉β r₁ r₂) (⇉a (⇉λ s₁) s₂) with ⇉-lc r₁ s₁ | ⇉-lc r₂ s₂
⇉-lc (⇉β r₁ r₂) (⇉a (⇉λ s₁) s₂) | w₁ , r₁' , s₁' | w₂ , r₂' , s₂' =
  w₁ [ w₂ / 0 ] , ⇉-sub 0 r₁' r₂' , ⇉β s₁' s₂'
⇉-lc (⇉a (⇉λ r₁) r₂) (⇉β s₁ s₂) with ⇉-lc r₁ s₁ | ⇉-lc r₂ s₂
... | w₁ , r₁' , s₁' | w₂ , r₂' , s₂' =
  w₁ [ w₂ / 0 ] , ⇉β r₁' r₂' , ⇉-sub 0 s₁' s₂'
```

```
⇉-lc (⇉a r₁ r₂) (⇉a s₁ s₂) with ⇉-lc r₁ s₁ | ⇉-lc r₂ s₂
... | (w₁ , r₁' , s₁') | (w₂ , r₂' , s₂') =
  w₁ · w₂ , ⇉a r₁' r₂' , ⇉a s₁' s₂'
⇉-lc (⇉λ r)      (⇉λ s) with ⇉-lc r s
⇉-lc (⇉λ r)      (⇉λ s) | w , r' , s' = λ w , (⇉λ r') , (⇉λ s')
```

Apart from recursive calls and the definition of parallel β-reduction, this proof uses the lemma ⇉-sub which states that parallel β-reduction is compatible with substitution: if t reduces to t' and u to u' then $t[u/x]$ reduces to $t'[u'/x]$. The proof follows the one of lemma 3.4.3.4:

```
⇉-sub : {t t' u u' : Tm} (x : ℕ) → t ⇉ t' → u ⇉ u' →
        t [ u / x ] ⇉ t' [ u' / x ]

⇉-sub x (⇉v y)        ru with x ≟ y
⇉-sub x (⇉v y)        ru | yes p = ru
⇉-sub x (⇉v y)        ru | no ¬p = ⇉v (↓ x y ¬p)
⇉-sub x (⇉a rt₁ rt₂) ru = ⇉a (⇉-sub x rt₁ ru) (⇉-sub x rt₂ ru)
⇉-sub x (⇉λ rt)       ru = ⇉λ (⇉-sub (suc x) rt (⇉-wk 0 ru))
⇉-sub x (⇉β {t} {t'} {u} {u'} rt₁ rt₂) ru =
  subst₂ _⇉_ refl
    (sym (sub-sub t' u' _ 0 x z≤n))
    (⇉β (⇉-sub (suc x) rt₁ (⇉-wk 0 ru)) (⇉-sub x rt₂ ru))
```

This function itself uses two auxiliary lemmas. The first one states that reduction is compatible with weakening:

```
⇉-wk : {t t' : Tm} (x : ℕ) → t ⇉ t' → wk x t ⇉ wk x t'
```

and the second one is a form of commutation for double substitution:

```
sub-sub : ∀ t u v x y → x ≤ y →
  t [ u / x ] [ v / y ] ≡ t [ wk x v / suc y ] [ u [ v / y ] / x ]
```

The latest requires considering a large number of cases depending on the relative values of x and y, and showing quite a few lemmas which were left behind the curtain in section 3.4.3:

– commutation of liftings: when $x \leqslant y$,

$$\uparrow_x \uparrow_y z = \uparrow_{y+1} \uparrow_x z$$

– commutation of unliftings: when $x \geqslant y$,

$$\downarrow_x \downarrow_y z = \downarrow_y \downarrow_{x+1} z$$

– commutation of liftings and unliftings:

$$\uparrow_x \downarrow_y z = \begin{cases} \downarrow_y \uparrow_{x+1} z & \text{when } x \geqslant y, \\ \downarrow_{y+1} \uparrow_x z & \text{when } x \leqslant y, \end{cases}$$

– commutation of weakenings: when $x \leqslant y$,

$$\uparrow_x \uparrow_y t = \uparrow_{y+1} \uparrow_x t$$

– commutation of weakening and substitution:

$$\uparrow_y(t[u/x]) = \begin{cases} (\uparrow_{y+1} t)[\uparrow_y u/x] & \text{when } x \leqslant y, \\ (\uparrow_y t)[\uparrow_y u/x + 1] & \text{when } x \geqslant y, \end{cases}$$

and

$$(\uparrow_x t)[u/x] = t$$

Details are left to the reader (and beware, they are of a quite combinatorial nature).

Confluence of the parallel β-reduction. In order to deduce that the parallel β-reduction is confluent, we first need to define the relation ⇉* as the reflexive and transitive closure of the parallel β-reduction relation ⇉:

```
_⇉*_ : Tm → Tm → Set
_⇉*_ = Star _⇉_
```

We can formally show lemma 3.4.3.6, stating that parallel β-reduction satisfies a property between local confluence and confluence, by

```
⇉-slconfl : {t u v : Tm} →
        t ⇉ u → t ⇉* v → Σ Tm (λ w → u ⇉* w × v ⇉ w)

⇉-slconfl {t} {u} {v} r ε = u , ε , r
⇉-slconfl r (s ◅ ss) with ⇉-lc r s
... | w' , s' , r' with ⇉-slconfl r' ss
... | w , ss' , r'' = w , s' ◅ ss' , r''
```

and deduce the confluence of the parallel β-reduction as in theorem 3.4.3.7 by

```
⇉-confl : {t u v : Tm} →
        t ⇉* u → t ⇉* v → Σ Tm (λ w → u ⇉* w × v ⇉* w)

⇉-confl {t} {u} {v} ε ss = v , ss , ε
⇉-confl {t} {u} {v} (r ◅ rr) ss with ⇉-slconfl r ss
... | w' , ss' , r' with ⇉-confl rr ss'
... | w , ss'' , rr' = w , ss'' , r' ◅ rr'
```

Confluence of the β-reduction. We can finally deduce the confluence of β-reduction, following the proof presented in sections 3.4 and 3.4.3. We first define the relation ↝* as the reflexive and transitive closure of the β-reduction relation ↝:

```
_↝*_ : Tm → Tm → Set
_↝*_ = Star _↝_
```

We can show that if a term *t* β-reduces to a term *u* then *t* parallel β-reduces to *u* (lemma 3.4.3.1):

```
↝→⇉ : {t u : Tm} → t ↝ u → t ⇉ u
↝→⇉ ↝β      = ⇉β ⇉-refl ⇉-refl
↝→⇉ (↝l r) = ⇉a (↝→⇉ r) ⇉-refl
↝→⇉ (↝r r) = ⇉a ⇉-refl (↝→⇉ r)
↝→⇉ (↝λ r) = ⇉λ (↝→⇉ r)
```

where the reflexivity of parallel β-reduction (lemma 3.4.2.1) is shown with

```
⇉-refl : {t : Tm} → t ⇉ t
⇉-refl {var x}  = ⇉v x
⇉-refl {t · t'} = ⇉a ⇉-refl ⇉-refl
⇉-refl {λ t}    = ⇉λ ⇉-refl
```

From there, we can easily show that iterated β-reduction implies iterated parallel β-reduction:

```
↠*→⇉* : {t u : Tm} → t ↠* u → t ⇉* u
↠*→⇉* ε         = ε
↠*→⇉* (r ◁ rr) = ↠→⇉ r ◁ ↠*→⇉* rr
```

Conversely, we can show that iterated parallel β-reduction implies iterated β-reduction (see lemma 3.4.3.3, formal proof is left to the reader):

```
⇉*→↠* : {t u : Tm} → t ⇉* u → t ↠* u
```

We can finally use this to deduce the confluence β-reduction (theorem 3.4.4.1) from the one of parallel β-reduction shown above:

```
↠-confl : {t u v : Tm} →
          t ↠* u → t ↠* v → Σ Tm (λ w → u ↠* w × v ↠* w)
↠-confl rr ss with ⇉-confl (↠*→⇉* rr) (↠*→⇉* ss)
... | w , ss' , rr' = w , ⇉*→↠* ss' , ⇉*→↠* rr'
```

7.4 Combinatory logic

Combinatory logic, which was presented in section 3.6.3, can be implemented in a way similar to pure λ-calculus. We begin by describing the type CL of combinatory logic terms:

```
data CL : Set where
  var  : ℕ → CL
  _·_  : CL → CL → CL
  S K I : CL
```

A term is thus either a variable, an application of a term to another, or one of the combinators S, K or I. Reduction of terms can then be formalized as a binary inductive predicate with constructors expressing the reduction rules for the combinators, as well as the fact that it is compatible with composition:

```
data _⇒_ : CL → CL → Set where
  ⇒-S : {T U V : CL}  →  S · T · U · V ⇒ (T · V) · (U · V)
  ⇒-K : {T U : CL}    →     K · T · U ⇒ T
  ⇒-I : {T : CL}      →         I · T ⇒ T
  ⇒-l : {T T' U : CL} → T ⇒ T' → T · U ⇒ T' · U
  ⇒-r : {T U U' : CL} → U ⇒ U' → T · U ⇒ T · U'
```

Abstraction. Following the definition given in section 3.6.3, we can define an analogue of abstraction for combinatory logic terms, which takes as argument a variable and a term in which the variable should be abstracted:

```
abs : ℕ → CL → CL
abs x (var y) with x ≟ y
abs x (var _) | yes p = I
abs x (var y) | no ¬p = K · var (↓ x y ¬p)
abs x (T · T')         = S · abs x T · abs x T'
abs x S               = K · S
abs x K               = K · K
abs x I               = K · I
```

Our aim is now to show that this is a reasonable notion of abstraction. In order to do so, we first define the substitution in a term T of a variable x by a term U:

```
_[_=:_] : CL → CL → ℕ → CL
var y [ U =: x ] with x ≟ y
(var y [ U =: _ ]) | yes p = U
(var y [ U =: x ]) | no ¬p = var (↓ x y ¬p)
(T · T') [ U =: x ]         = (T [ U =: x ]) · (T' [ U =: x ])
S [ U =: x ]               = S
K [ U =: x ]               = K
I [ U =: x ]               = I
```

We also need to consider the reflexive and transitive closure ⇨* of the reduction relation ⇨ by

```
_⇨*_  : CL → CL → Set
_⇨*_ = Star _⇨_
```

Finally, we can formalize lemma 3.6.3.5, which states that abstraction behaves as expected: $(\Lambda x.T)\, U$ reduces to $T[U/x]$.

```
cl-β : (x : ℕ) (T U : CL) → (abs x T) · U ⇨* T [ U =: x ]
cl-β x (var y) U with x ≟ y
cl-β x (var _) U | yes p = ⇨-I ◁ ε
cl-β x (var y) U | no ¬p = ⇨-K ◁ ε
cl-β x (T · U) V         = ⇨-S ◁ ⇨*-· (cl-β x T V) (cl-β x U V)
cl-β x S       U         = ⇨-K ◁ ε
cl-β x K       U         = ⇨-K ◁ ε
cl-β x I       U         = ⇨-K ◁ ε
```

This proof uses the auxiliary lemma

```
⇨*-·  : {T T' U U' : CL} → T ⇨* T' → U ⇨* U' → T · U ⇨* T' · U'
```

which states that reduction is compatible with concatenation and whose proof is left to the reader.

Exercise 7.4.0.1. Formalize the translations between λ-terms and combinatory logic terms of section 3.6.3, i.e. define functions

```
icl : Tm → CL
icl (var x)  = var x
icl (t · t') = icl t · icl t'
icl (ƛ t)    = abs zero (icl t)
```

and

```
ilam : CL → Tm
ilam (var x)  = var x
ilam (T · T') = ilam T · ilam T'
ilam S        = ƛ ƛ ƛ (var 2 · var 0 · (var 1 · var 0))
ilam K        = ƛ ƛ var (suc 0)
ilam I        = ƛ (var 0)
```

and show various lemmas expressing preservation of reduction such as lemma 3.6.3.7:

```
ilam-red : {T U : CL} → T ⇒ U → ilam T ↠* ilam U
```

7.5 Simply typed λ-calculus

7.5.1 Definition. We now present a formalization of simply typed λ-calculus introduced in chapter 4. The reader is strongly advised to try this by himself before reading the section: what is easy to read is not necessarily easy to write! This is inspired by the excellent course [WK19].

Types. We suppose fixed an infinite countable set of type variables, say the natural numbers:

```
TVar : Set
TVar = ℕ
```

and the types are inductively to be defined as type variables of arrows between types:

```
data Type : Set where
  X   : TVar → Type
  _⇒_ : Type → Type → Type
```

Contexts. A context is simply a list of types. However, in order to adopt the usual notations, instead of defining the type `Ctxt` of context as `List Type`, we use the following definition:

```
data Ctxt : Set where
  ∅   : Ctxt
  _,_ : Ctxt → Type → Ctxt
```

a context is thus either the empty context `∅` or of the form `Γ , A` for some context `Γ` and type `A`.

Terms. The type `Γ ⊢ A` of terms of type `A` in the context `Γ` is defined by induction by

```
data _⊢_ : Ctxt → Type → Set where
  var : ∀ {Γ A}   →                    Γ ∋ A → Γ ⊢ A
  _·_ : ∀ {Γ A B} → Γ ⊢ (A ⇒ B) → Γ ⊢ A → Γ ⊢ B
  ƛ_  : ∀ {Γ A B} →              Γ , A ⊢ B → Γ ⊢ A ⇒ B
```

where the constructors of the inductive type correspond to the typing rules of simply typed λ-calculus given in section 4.1.4. In this formalization, we are right in the middle of the Curry-Howard correspondence: a proof that $\Gamma \vdash A$ is derivable is precisely a λ-term t of type A in the context Γ. In the constructor corresponding to variables, we use $\Gamma \ni A$, which is the type of proofs that a type A belongs to Γ: such a proof essentially consists of a natural number n such that the n-th element of Γ is A. Formally, it can be defined as follows:

```
data _∍_ : Ctxt → Type → Set where
  zero : ∀ {Γ A}           → (Γ , A) ∍ A
  suc  : ∀ {Γ B A} → Γ ∍ A → (Γ , B) ∍ A
```

Note that this corresponds to identifying variables by their de Bruijn index in the context.

Weakening. In order to define substitution and β-reduction, we have to make use of the weakening rule

$$\frac{\Gamma \vdash t : A}{\Gamma, x : B \vdash t : A} \; (\text{wk})$$

and thus need to show that this rule is admissible. A naive approach would consist in trying to show the following corresponding lemma:

```
wk : ∀ {Γ A B} → Γ ⊢ A → Γ , B ⊢ A
```

However, we cannot manage to prove it because the induction hypothesis is not strong enough in the case of abstraction: we have to show

$$\frac{\Gamma, x : B, y : A \vdash t : A'}{\Gamma, x : B \vdash \lambda y^A.t : A \to A'} \; (\text{wk})$$

and we cannot use the induction hypothesis on the premise because the weakened variable x is not in the last position in the context. In order to overcome this problem, we could prove the following generalization of the weakening rule:

$$\frac{\Gamma, \Delta \vdash t : A}{\Gamma, x : B, \Delta \vdash t : A} \; (\text{wk})$$

It will turn out equally easy and more natural to prove the following even more general version:

$$\frac{\Gamma \vdash t : A}{\Delta \vdash t : A} \; (\text{wk})$$

whenever Γ is obtained from Δ by removing multiple typed variables, which we write $\Gamma \subseteq \Delta$ (this corresponds to performing at once multiple weakening rules in the previous version). We thus define the "inclusion" relation between contexts as

```
data _⊆_ : Ctxt → Ctxt → Set where
  ∅⊆∅  : ∅ ⊆ ∅
  keep : ∀ {Γ Δ A} → Γ ⊆ Δ → Γ , A ⊆ Δ , A
  drop : ∀ {Γ Δ A} → Γ ⊆ Δ →     Γ ⊆ Δ , A
```

and prove weakening as follows

```
wk : ∀ {Γ Δ A} → Γ ⊆ Δ → Γ ⊢ A → Δ ⊢ A
wk i (var x)  = var (wk-var i x)
wk i (t · t') = wk i t · wk i t'
wk i (ƛ t)    = ƛ wk (keep i) t
```

where, in the case of variables, we use the following lemma showing that if a type belongs to a context, it still belongs to it if we add types to this context:

```
wk-var : ∀ {Γ Δ A} → Γ ⊆ Δ → Γ ∋ A → Δ ∋ A
wk-var (keep i) zero    = zero
wk-var (keep i) (suc x) = suc (wk-var i x)
wk-var (drop i) x       = suc (wk-var i x)
```

Finally, we can show that the first weakening rule considered above can be deduced as the particular cases where the inclusion is of the form $\Gamma \subseteq \Gamma, A$:

```
wk-last : ∀ {Γ A B} → Γ ⊢ A → Γ , B ⊢ A
wk-last t = wk (drop ⊆-refl) t
```

where `⊆-refl` is a proof that inclusion is reflexive:

```
⊆-refl : ∀ {Γ} → Γ ⊆ Γ
⊆-refl {∅}     = ∅⊆∅
⊆-refl {Γ , A} = keep ⊆-refl
```

Substitution. We can define substitution as follows. Given a term t in a context Γ, a variable x in the context Γ and a term u of the right type, we want to construct a term $t[u/x]$ obtained by replacing all occurrences of x by u in t. It turns out to be simpler to define a generalization of this operation, and replace all the variables of Γ at once in t: given a function σ (a *substitution*) which to a variable of Γ associates a term of appropriate type, we define the term $t[\sigma]$ obtained from t by replacing every free variable x by $\sigma(x)$ as follows.

```
_[_] : ∀ {Γ Δ A} → Γ ⊢ A → (∀ {B} → Γ ∋ B → Δ ⊢ B) → Δ ⊢ A
var x    [ σ ] = σ x
(t · t') [ σ ] = (t [ σ ]) · (t' [ σ ])
(ƛ t)    [ σ ] = ƛ (t [ (λ { zero → var zero ;
                            (suc x) → wk-last (σ x) }) ])
```

In order to define β-reduction, we will only be interested in substituting the last variable of the context, which can of course be recovered as a particular case:

```
_[_/0] : ∀ {Γ A B} → Γ , B ⊢ A → Γ ⊢ B → Γ ⊢ A
t [ u /0] = t [ (λ { zero → u ; (suc x) → var x }) ]
```

β-reduction. The β-reduction can then be defined as follows, similarly to the case of untyped λ-calculus:

```
data _↝_ {Γ : Ctxt} : {A : Type} → Γ ⊢ A → Γ ⊢ A → Set where
  ↝β : ∀ {A B} (t : Γ , A ⊢ B) (u : Γ ⊢ A) →
       (ƛ t) · u ↝ t [ u /0]
```

```
↝l : ∀ {A B} {t t' : Γ ⊢ A ⇒ B} → t ↝ t' → (u : Γ ⊢ A) →
     t · u ↝ t' · u
↝r : ∀ {A B} (t : Γ ⊢ A ⇒ B) → {u u' : Γ ⊢ A} → u ↝ u' →
     t · u ↝ t · u'
↝λ : ∀ {A B} {t t' : Γ , A ⊢ B} → t ↝ t' →
     λ t ↝ λ t'
```

where we use substitution in the first case, as indicated above.

7.5.2 Strong normalization. In order to show the effectiveness of the implementation performed in the previous section, we shall prove a major theorem of λ-calculus: the strong normalization theorem (theorem 4.2.2.5) which states that every typable term is strongly normalizing. We follow here the proof given in section 4.2.2 using reducibility candidates. Similar proofs in Coq can be found in [PdAC+10].

Strong normalizability. We first have to define what it means for the reduction relation ↝ to be *halting*, or *strongly normalizing*. A term t is halting when there is no infinite reduction starting from it. It is however generally a bad idea to define concepts by negation, because we lose constructivity, and we will not directly adopt this definition. Instead, we will define by induction that a term t is halting whenever all the terms it can reduce to are themselves halting:

```
data halts {Γ : Ctxt} {A : Type} : Γ ⊢ A → Set where
  sn : {t : Γ ⊢ A} → ({t' : Γ ⊢ A} → t ↝ t' → halts t') → halts t
```

Note that we could have equivalently defined t to be halting when it is accessible (see section 6.8.6) with respect to the opposite of the reduction relation:

```
halts : ∀ {Γ A} → Γ ⊢ A → Set
halts t = Acc _↩_ t
```

where the opposite of the reduction relation is

```
_↩_ : ∀ {Γ A} → Γ ⊢ A → Γ ⊢ A → Set
u ↩ t = t ↝ u
```

Induction on the reduction. We can define the iterated reduction relation as usual by

```
_↝*_ : ∀ {Γ A} → Γ ⊢ A → Γ ⊢ A → Set
_↝*_ = Star _↝_
```

Given a halting term t, the reduction relation ↝ is terminating on terms u such that t ↝* u. We can thus reason by well-founded induction on it, i.e. we have the following induction principle:

```
↝-rec : ∀ {Γ A} {t : Γ ⊢ A} → halts t → (P : Γ ⊢ A → Set) →
        ({u : Γ ⊢ A} → t ↝* u → ({v : Γ ⊢ A} → u ↝ v → P v) → P u) →
        {u : Γ ⊢ A} → t ↝* u → P u
```

whose proof is left to the reader.

Reducibility candidates. We formalize here the "typed" variant of reducibility candidates discussed in remark 4.2.2.6. We define sets $R_{\Gamma \vdash A}$, indexed by contexts Γ and types A, consisting of terms of type A in the context Γ, by induction on the type A by

```
R : {Γ : Ctxt} {A : Type} (t : Γ ⊢ A) → Set
R {Γ} {X _}   t = halts t
R {Γ} {A ⇒ B} t = {u : Γ ⊢ A} → R u → R (t · u)
```

The core of the proof then consists in showing the three properties of proposition 4.2.2.1 satisfied by reducibility candidates:

```
CR1 : ∀ {Γ A} {t     : Γ ⊢ A} →
      R t → halts t
CR2 : ∀ {Γ A} {t t' : Γ ⊢ A} →
      R t → t ↝ t' → R t'
CR3 : ∀ {Γ A} {t     : Γ ⊢ A} →
      neutral t → ({t' : Γ ⊢ A} → t ↝ t' → R t') → R t
```

where neutral terms are characterized by the following predicate:

```
data neutral : ∀ {Γ A} → Γ ⊢ A → Set where
  nvar : ∀ {Γ A} (x : Γ ∋ A) → neutral (var x)
  napp : ∀ {Γ A B} (t : Γ ⊢ A ⇒ B) (u : Γ ⊢ A) → neutral (t · u)
```

As in the proof of the above proposition, we show all three properties together and reason by induction on the type A. The formal proof is shown below:

```
CR1 {Γ} {X _}   r = r
CR1 {Γ} {A ⇒ B} {t} r =
  halts-vapp t x? (CR1 (r (CR3 (nvar x?) (λ ()))))
CR2 {Γ} {X _}     (sn f) b = f b
CR2 {Γ} {A ⇒ B} r b {u} Ru = CR2 (r Ru) (↝l b u)
CR3 {Γ} {X _}   n f = sn f
CR3 {Γ} {A ⇒ B} {t} n f {u} Ru = lem u ε
  where
  CR2* : {t t' : Γ ⊢ A} → t ↝* t' → R t → R t'
  CR2* ε Rt = Rt
  CR2* {t} {t'} (b ◂ bb) Rt = CR2* bb (CR2 Rt b)
  lem : ∀ v → u ↝* v → R (t · v)
  lem v u↝*v = ↝-rec (CR1 Ru) (λ v → R (t · v))
    (λ {w} u↝*w ind →
      CR3 (napp t w)
        λ { (↝l t↝t' u) → f t↝t' (CR2* u↝*w Ru) ;
            (↝r t w↝w') → ind w↝w' }
    ) u↝*v
```

We do not detail it, because it follows closely the proof of proposition 4.2.2.1, except for two points in the second case of CR1.

   We recall that this part of the proof consists in showing that for every term $t \in R_{\Gamma \vdash A \to B}$, we have that t is halting and goes on as follows. Consider a variable x such that $\Gamma \vdash x : A$ is derivable: this variable is neutral and thus in R_A by (CR3), therefore $t\,x$ belongs to $R_{\Gamma \vdash B}$ is thus halting, from which

we deduce that t must also be halting. The last step is taken care of by the lemma `halts-vapp`, which states that if the term $t\,x$ is terminating then t is also terminating:

```
halts-vapp : ∀ {Γ A B} (t : Γ ⊢ A ⇒ B) → (x : Γ ∋ A) →
              halts (t · var x) → halts t
```

The (easy) proof is left to the reader. We have to confess that we have been cheating in the above proof: there is no reason that we should have a variable x such that $\Gamma \vdash x : A$, unless A belongs to Γ, which nothing guarantees here (this was not a problem in the proof of proposition 4.2.2.1, because we were working in an untyped setting). In our Agda proof, we have simply been postulating the existence of such a variable:

```
postulate x? : ∀ {Γ A} → Γ ∋ A
```

Of course, this is wrong, but it can be mitigated in two ways. First, if we had a more full-fledged programming language with data types (natural numbers, booleans, strings, etc.), we could prove that every program of a type which does not contain type variables is terminating in the same way, by using values instead of variables, and this would cover most cases of interest. For instance, supposing that we have a type $\mathbb{N}$ of natural numbers, we can show that $t \in R_{\mathbb{N}\to\mathbb{N}}$ because, by induction hypothesis, we have that $t\,5$ is terminating and reason as above. Another way to solve the problem is to change slightly the proof of the second case of CR1. Suppose that $t \in R_{\Gamma\vdash A}$, by weakening we have that $\Gamma, x : A \vdash t : A$, and now we have the variable x such that $\Gamma, x : A \vdash x : A$: by induction hypothesis we have that $t\,x$ is halting, therefore the weakening of t is terminating, and therefore t is terminating. In practice, this makes the proof much more delicate, because we have to explicitly deal with matters related to weakening: in Agda, the weakening of t is not the same as t. Moreover, the definition of reducibility candidates has to be slightly generalized in order to take weakening in account and have the right induction hypothesis [Sak14]:

```
R : {Γ : Ctxt} {A : Type} (t : Γ ⊢ A) → Set
R {Γ} {X _}   t = halts t
R {Γ} {A ⇒ B} t = {Γ' : Ctxt} {u : Γ' ⊢ A} →
                  (i : Γ ⊆ Γ') → R u → R (wk i t · u)
```

Strong normalization. Finally, we can deduce that simply typed λ-terms are strongly normalizing by following section 4.2.2. We do not detail the proofs here. Lemma 4.2.2.2 can be formalized as

```
R-abs : ∀ {Γ A B} (t : Γ , A ⊢ B) →
        ((u : Γ ⊢ A) → R (t [ u /0])) → R (ƛ t)
```

lemma 4.2.2.3 as

```
R-sub : ∀ {Γ A} (t : Γ ⊢ A)
        (σ : ∀ {B} → Γ ∋ B → Γ ⊢ B) →
        (∀ {B} → (x : Γ ∋ B) → R (σ x)) → R (t [ σ ])
```

the adequacy proposition 4.2.2.4 as

```
R-all : ∀ {Γ A} (t : Γ ⊢ A) → R t
```

and finally the strong normalization theorem 4.2.2.5 as

```
SN : ∀ {Γ A} (t : Γ ⊢ A) → halts t
SN t = CR1 (R-all t)
```

Exercise 7.5.2.1. Extend the language of section 7.1 adding abstractions and show preservation of types, progress and strong normalization.

Weak normalization. In a typical study of a programming language, one is not usually interested in showing that every possible way of reducing programs terminates, but only that this is the case with the particular reduction strategy used by the language. In this case, the above proof can be somewhat simplified, as explained in section 4.2.5, which we now illustrate by showing that the call-by-value reduction strategy terminates for the simply typed λ-calculus.

Following section 3.5.1, we can characterize values and neutral terms by

```
data value   : ∀ {Γ A} (t : Γ ⊢ A) → Set
data neutral : ∀ {Γ A} (t : Γ ⊢ A) → Set

data value where
  vabs : ∀ {Γ} {A B} (t : Γ , A ⊢ B) → value (ƛ t)
  vneu : ∀ {Γ A} {t : Γ ⊢ A} → neutral t → value t
data neutral where
  nvar : ∀ {Γ A} (x : Γ ∋ A) → neutral (var x)
  napp : ∀ {Γ A B} {t : Γ ⊢ A ⇒ B} {u : Γ ⊢ A} →
         neutral t → value u → neutral (t · u)
```

the call-by-value reduction strategy is

```
data _↝_ : ∀ {Γ A} → Γ ⊢ A → Γ ⊢ A → Set where
  ↝β : ∀ {Γ A B} (t : Γ , A ⊢ B) → {u : Γ ⊢ A} →
       value u → (ƛ t) · u ↝ (t [ u /0])
  ↝l : ∀ {Γ A B} {t t' : Γ ⊢ A ⇒ B} →
       t ↝ t' → (u : Γ ⊢ A) → t · u ↝ t' · u
  ↝r : ∀ {Γ A B} {t : Γ ⊢ A ⇒ B} {u u' : Γ ⊢ A} →
       value t → u ↝ u' → t · u ↝ t · u'
```

and the iterated reduction ↝* is defined as usual. It is not hard to show that values and neutral terms are normal forms

```
value-nf   : ∀ {Γ A} {t t' : Γ ⊢ A} → value   t → ¬ (t ↝ t')
neutral-nf : ∀ {Γ A} {t t' : Γ ⊢ A} → neutral t → ¬ (t ↝ t')
value-nf   (vneu (napp n v)) (↝l r u) = neutral-nf n r
value-nf   (vneu (napp n v)) (↝r t r) = value-nf   v r
neutral-nf (napp n v)        (↝l r u) = neutral-nf n r
neutral-nf (napp n v)        (↝r t r) = value-nf   v r
```

and that the reduction is deterministic

```
det : ∀ {Γ A} {t t₁ t₂ : Γ ⊢ A} → t ↝ t₁ → t ↝ t₂ → t₁ ≡ t₂
det (↝β t u₁) (↝β .t u₂) = refl
det (↝β t u)  (↝r t' r)  = ⊥-elim (value-nf u r)
det (↝l r₁ u) (↝l r₂ .u) = cong₂ _·_ (det r₁ r₂) refl
```

```
det (↬l r₁ u) (↬r t r₂)  = ⊥-elim (value-nf t r₁)
det (↬r t r₁) (↬β t₁ u)  = ⊥-elim (value-nf u r₁)
det (↬r t r₁) (↬l r₂ u)  = ⊥-elim (value-nf t r₂)
det (↬r t r₁) (↬r x r₂)  = cong₂ _·_ refl (det r₁ r₂)
```

The definition of the predicate `halts` is the same as above and one can easily show that it is preserved under reduction:

```
↬-halts : ∀ {Γ A} {t t' : Γ ⊢ A} → t ↬ t' → halts t → halts t'
↬-halts r (sn h) = h r
```

In fact this could also be shown with the general β-reduction. The novelty brought by using call-by-value reduction is that it is deterministic, and we thus now also have the "converse" property:

```
↫-halts : ∀ {Γ A} {t t' : Γ ⊢ A} → t ↬ t' → halts t' → halts t
↫-halts r h = sn (λ r' → subst halts (det r r') h)
```

Finally, the induction principle `↬-rec` presented for β-reduction of course still holds with call-by-value reduction.

We take the following variant of the definition of reducibility candidates (note that we suppose that `t` is halting in both cases):

```
R : {Γ : Ctxt} {A : Type} (t : Γ ⊢ A) → Set
R {Γ} {X _}    t = halts t
R {Γ} {A ⇒ B} t = halts t × ({u : Γ ⊢ A} → R u → R (t · u))
```

The three properties of candidates can now be proved independently and in a much simpler way (in particular, we do not need the `x?` hack that we used above):

```
CR1 : ∀ {Γ A} {t : Γ ⊢ A} → R t → halts t
CR1 {Γ} {X x} r = r
CR1 {Γ} {A ⇒ B} {t} r = fst r

CR2 : ∀ {Γ A} {t t' : Γ ⊢ A} → R t → t ↬ t' → R t'
CR2 {Γ} {X x} r b = ↬-halts b r
CR2 {Γ} {A ⇒ B} {t} r b =
  ↬-halts b (fst r) , (λ {u} Ru → CR2 (snd r Ru) (↬l b u))

CR3 : ∀ {Γ A} {t t' : Γ ⊢ A} → t ↬ t' → R t' → R t
CR3 {Γ} {X x} b r = ↫-halts b r
CR3 {Γ} {A ⇒ A₁} b r =
  (↫-halts b (fst r)) , (λ {u} Ru → CR3 (↬l b u) (snd r Ru))
```

7.5.3 Normalization by evaluation. We finally present a way to compute the normal form of terms in simply typed λ-calculus using normalization by evaluation, see section 3.5.2. The idea is that we are going to interpret λ-terms as Agda functions on normal forms, so that we can use Agda's built-in reduction mechanism, and then translate the result back to λ-calculus. More precisely, given a type A, we write $[\![A]\!]$ for the set defined inductively by

– $[\![X]\!] = \mathrm{NF}_A$: the set associated to a type variable is the set of all terms of type A in normal form,

– $[\![A \to B]\!] = [\![A]\!] \to [\![B]\!]$: the set associated to the arrow type $A \to B$ is the set of all functions from $[\![A]\!]$ to $[\![B]\!]$.

Then, we are going to interpret a term t of type A as an element $[\![t]\!]_\rho$ of $[\![A]\!]$. Since we need to properly take care of the free variables of t, our interpretation also depends on an environment ρ, which is is a function which to every free variable of t associates a term in normal form. The interpretation is defined by induction on t by

$$[\![x]\!]_\rho = \rho(x) \qquad [\![\lambda x.t]\!] = u \mapsto [\![t]\!]_{(\rho, x \mapsto u)} \qquad [\![t\, u]\!] = [\![t]\!]_\rho([\![u]\!]_\rho)$$

(above, $(\rho, x \mapsto u)$ is the environment which behaves as ρ, except that it associates u to x). In other words, we evaluate the term t in the environment ρ. Finally, we are going to define, for every type A, a *reification* function $\downarrow_A$, which translates every element of the set $[\![A]\!]$ to a λ-term in normal form. The definition will of course be performed by induction on the type A. In order to handle the case where A is an arrow type, it turns out that we also need a *reflection* function $\uparrow_A$ which allows us to see a variable of type A as an element of $[\![A]\!]$. Actually, in order to be able to perform the definition of $\uparrow_A$ by induction, we need to define it on all neutral terms and not only variables, and define it together with reification. We thus define two functions

$$\downarrow_A : [\![A]\!] \to \mathrm{NF}_A \qquad\qquad \uparrow_A : \mathrm{NE}_A \to [\![A]\!]$$

where NF_A and NE_A are respectively the normal forms and neutral terms of type A, by induction on A by:

$$\downarrow_X t = t \qquad\qquad \uparrow_X t = t$$
$$\downarrow_{A \to B} f = \lambda x.\, \downarrow_B f(\uparrow_A x) \qquad\qquad \uparrow_{A \to B} t = u \mapsto \uparrow_B (t\,(\downarrow_A u))$$

where x is supposed to be "fresh" in the lower left case. Finally, we can compute the normal form $\hat{t}$ of a λ-term t by

$$\hat{t} = \downarrow_A [\![t]\!]_{\rho_0}$$

where ρ_0 is the "trivial environment" which to a variable x associates the variable x.

Terms. Our actual formalization is inspired by [Arn17]. We use the same definitions as above for types, contexts, and λ-terms. Inspired by the notation for bidirectional typechecking (section 4.4.5), we write $\Gamma \hookleftarrow A$ (resp. $\Gamma \rightarrowtail A$) for the type of *normal forms* (resp. *neutral terms*) of type A in the context Γ, defined as the following inductive types:

```
data _↤_ : Ctxt → Type → Set
data _↦_ : Ctxt → Type → Set

data _↤_ where
  abs : ∀ {Γ A B} → Γ , A ↤ B → Γ ↤ A ⇒ B
  neu : ∀ {Γ A}   →     Γ ↦ A → Γ ↤ A

data _↦_ where
  var : ∀ {Γ A}   →             Γ ∋ A → Γ ↦ A
  app : ∀ {Γ A B} → Γ ↦ A ⇒ B → Γ ↤ A → Γ ↦ B
```

Note that those are not characterized here by a predicate on terms as before, but rather implemented as a new inductive type. For this reason, we need to implement again substitution on those types:

```
_↩[_] : ∀ {Γ Δ A} → Γ ↩ A → Γ ⊆ Δ → Δ ↩ A
_↦[_] : ∀ {Γ Δ A} → Γ ↦ A → Γ ⊆ Δ → Δ ↦ A
abs t   ↩[ σ ] = abs (t ↩[ keep σ ])
neu t   ↩[ σ ] = neu (t ↦[ σ ])
var x   ↦[ σ ] = var (x v[ σ ])
app t u ↦[ σ ] = app (t ↦[ σ ]) (u ↩[ σ ])
```

where the case of variables is handled by

```
_v[_] : ∀ {Γ Δ A} → Γ ∋ A → Γ ⊆ Δ → Δ ∋ A
zero  v[ keep σ ] = zero
suc x v[ keep σ ] = suc (x v[ σ ])
x     v[ drop σ ] = suc (x v[ σ ])
```

Interpreting types. The interpretation of types as sets of terms is performed following the above definition. We actually need this definition to also depend on a context and write ⟦ Γ ⊢ A ⟧ for the interpretation of the type A in the context Γ, which is defined by

```
⟦_⊢_⟧ : Ctxt → Type → Set
⟦ Γ ⊢ X i ⟧   = Γ ↦ X i
⟦ Γ ⊢ A ⇒ B ⟧ = ∀ {Δ} → Γ ⊆ Δ → ⟦ Δ ⊢ A ⟧ → ⟦ Δ ⊢ B ⟧
```

In the second case, we need to incorporate the weakening of context in the definition in order to be able to produce "fresh variables" when reifying functions.

Reflection and reification. The reflection and reification functions are defined by mutual induction by following their definition given above. We also need to define a function Var which is the variable corresponding to the last element of the context in the set ⟦Γ ⊢ A⟧:

```
Var : ∀ {Γ A} → ⟦ Γ , A ⊢ A ⟧
↑ :   ∀ {Γ A} →   Γ ↦ A   → ⟦ Γ ⊢ A ⟧
↓ :   ∀ {Γ A} → ⟦ Γ ⊢ A ⟧ →   Γ ↩ A

Var {Γ} {X i}       = var zero
Var {Γ} {A ⇒ B} σ t = ↑ ((var zero) ↦[ σ ]) ⊆-refl t

↑ {Γ} {X i} t       = t
↑ {Γ} {A ⇒ B} t σ u = ↑ (app (t ↦[ σ ]) (↓ u))

↓ {Γ} {X i} t       = neu t
↓ {Γ} {A ⇒ B} f     = abs (↓ (f (drop ⊆-refl) Var))
```

Interpreting terms. We finally need to define the interpretation ⟦t⟧ρ of terms t, for which we first need to introduce the notion of environment, which will be done in a typeful fashion here. Given contexts Γ and Δ, we write ⟦ Δ ⊢⋆ Γ ⟧ for type of environment which to every variable of type A in Γ associates a normal form of type A in Δ:

```
⟦_⊢*_⟧ : Ctxt → Ctxt → Set
⟦ Δ ⊢* Γ ⟧ = ∀ {A} (x : Γ ∋ A) → ⟦ Δ ⊢ A ⟧
```

These are the environments adapted to terms whose free variables are in Γ. We can define the interpretation of terms following the above definition by

```
⟦_⟧ : ∀ {Γ Δ A} → Γ ⊢ A → ⟦ Δ ⊢* Γ ⟧ → ⟦ Δ ⊢ A ⟧
⟦ var x ⟧ ρ = ρ x
⟦ t · u ⟧ ρ = (⟦ t ⟧ ρ) ⊆-refl (⟦ u ⟧ ρ)
⟦ ƛ t ⟧   ρ = λ σ u → ⟦ t ⟧ ((λ x → wk* σ (ρ x)) ,* u)
```

In this definition, we have used the following auxiliary function, which extends an environment with a new value

```
_,*_ : ∀ {Γ Δ A} → ⟦ Δ ⊢* Γ ⟧ → ⟦ Δ ⊢ A ⟧ → ⟦ Δ ⊢* (Γ , A) ⟧
(ρ ,* t) zero    = t
(ρ ,* t) (suc x) = ρ x
```

as well as the following weakening principle for sets of normal forms:

```
wk* : ∀ {Γ Δ A} → Γ ⊆ Δ → ⟦ Γ ⊢ A ⟧ → ⟦ Δ ⊢ A ⟧
wk* {Γ} {Δ} {X i}   σ t = t ↦[ σ ]
wk* {Γ} {Δ} {A ⇒ B} σ f = λ τ t → f (⊆-trans σ τ) t
```

Computing normal forms. We can finally define the normalization of λ-terms by

```
normalize : ∀ {Γ A} → Γ ⊢ A → Γ ↤ A
normalize t = ↓ (⟦ t ⟧ id*)
```

where the trivial environment is

```
id* : ∀ {Γ} → ⟦ Γ ⊢* Γ ⟧
id* x = ↑ (var x)
```

An example. For instance, we can define the term $t = (\lambda x.\lambda y.x)\, x$ whose type is $x : X_0 \vdash t : X_1 \Rightarrow X_0$ by

```
K : ∅ , X 0 ⊢ X 0 ⇒ X 1 ⇒ X 0
K = ƛ (ƛ var (suc zero))

V : ∅ , X 0 ⊢ X 0
V = var zero

t : ∅ , X 0 ⊢ X 1 ⇒ X 0
t = K · V
```

If we ask Agda to compute (i.e. normalize) the normalized term `normalize t`, we obtain

```
abs (neu (var (suc zero)))
```

which is Agda's way of saying $\lambda y.x$, as expected.

Handling η-conversion. The above algorithm can be used in order to test whether two λ-terms are β-convertible: in order to know whether t and u are convertible, we simply need to look whether their respective normal forms $\hat{t}$ and $\hat{u}$ are equal. However this does not work if we want to test for βη-convertibility. For instance, the terms $t = \lambda x.\lambda y.x\,y$ and $u = \lambda x.x$, of type $(X \to Y) \to X \to Y$ are η-convertible and in normal form: we have $\hat{t} = t \neq u = \hat{u}$. In order to overcome this problem, a nice solution consists in slightly tightening the notion of normal form we consider, and require that terms with an arrow type should be abstractions: normal forms satisfying this are called *η-long normal forms.* In the definition of normal forms, this amounts to allowing considering neutral terms as normal ones, only when they have base types (and not arrow types), i.e. the definition of normal forms becomes

```
data _↵_ where
  abs : ∀ {Γ A B} → Γ , A ↵ B → Γ ↵ A ⇒ B
  neu : ∀ {Γ i}   →   Γ ↦ X i → Γ ↵ X i
```

Exercise 7.5.3.1. Modify the above normalization by evaluation algorithm in order to compute η-long normal forms.

Dependent type theory

We now introduce the logic we have seen at work in Agda. The type theory that we are presenting here was originally introduced by Martin-Löf in 1972 [ML75, ML82, ML98], most of Martin-Löf's work being freely accessible at [ML]. Its types are said to be *dependent* because they can depend on values. For instance, we can define a type Vec n of lists of length n, which depends on the natural number n. Another major feature of this type theory is that we can manipulate types as any other data: for instance, we can define functions which create types from other types, etc. In order to make this possible, the distinction between types and terms is dropped: types are simply the terms which admit a particular type, called "Type". Making all this work together nicely requires quite some care.

The core of the type theory is presented in section 8.1, universes being added in section 8.2, other usual type constructors in section 8.3 and inductive types in section 8.4. The ways a dependent proof assistant can be implemented is discussed in section 8.5

8.1 Core dependent type theory

In this section, we begin with the "minimal" version of dependent type theory, i.e. with (dependently typed) functions only. This is extended with more type constructors in section 8.3.

8.1.1 Expressions. As indicated above, there is no distinction between terms and types and we call them both "expressions" in order to make this clear. As usual, we suppose fixed an infinite countable supply of variables. An *expression e* is a term of the form

$$e, e' ::= x \mid e\,e' \mid \lambda x^e.e' \mid \Pi(x : e).e' \mid \text{Type}$$

In the following, we keep the old habit of writing t and A for expressions thought of as terms and as types, even though we cannot syntactically distinguish between both. The expressions can be read as follows:

– x: a term or a type variable,

– $t\,u$: application of a term to a term (or a type),

– $\lambda x^A.t$: the function (the λ-term) which to an element x of type A associates t,

– $\Pi(x : A).B$: the type of (dependent) functions from A to B,

– Type: the type of all types.

In Agda notation, $\Pi(x : A).B$ is written (x : A) → B and Type is written Set.

8.1.2 Free variables and substitution. In an expression of the form $\lambda x^A.t$ (resp. $\Pi(x : A).B$), the variable x is said to be *bound* in t (resp. in B), and expressions are considered modulo renaming of bound variables, which is called *α-equivalence*. A variable which is not bound is *free* and we write $\mathrm{FV}(e)$ for the set of free variables of an expression e, which is defined by

$$\mathrm{FV}(x) = \{x\}$$
$$\mathrm{FV}(t\,u) = \mathrm{FV}(t) \cup \mathrm{FV}(u)$$
$$\mathrm{FV}(\lambda x^A.t) = \mathrm{FV}(A) \cup (\mathrm{FV}(t) \setminus \{x\})$$
$$\mathrm{FV}(\Pi(x : A).B) = \mathrm{FV}(A) \cup (\mathrm{FV}(B) \setminus \{x\})$$
$$\mathrm{FV}(\mathrm{Type}) = \emptyset$$

We say that a variable x *occurs* in an expression A when $x \in \mathrm{FV}(A)$.

Given expressions e and u and a variable x, we define the *substitution $e[u/x]$* of x by u in e by induction on e:

$$\begin{aligned}
x[u/x] &= u & \\
y[u/x] &= y & \text{if } x \neq y \\
(t\,t')[u/x] &= (t[u/x])\,(t'[u/x]) & \\
(\lambda y^A.t)[u/x] &= \lambda y^{A[u/x]}.t[u/x] & \text{with } y \notin \mathrm{FV}(u) \cup \{x\} \\
(\Pi(y : A).B)[u/x] &= \Pi(y : A[u/x]).B[u/x] & \text{with } y \notin \mathrm{FV}(u) \cup \{x\} \\
\mathrm{Type}\,[u/x] &= \mathrm{Type} &
\end{aligned}$$

8.1.3 Contexts. A *context* Γ is a list

$$\Gamma = x_1 : A_1, \ldots, x_n : A_n$$

where the x_i are variables and the A_i are expressions. We sometimes write $\varnothing$ for the empty context, although we usually omit writing it. The set of free variables of a context is defined by

$$\mathrm{FV}(\Gamma) = \bigcup_{i=1}^{n} \mathrm{FV}(A_i)$$

and we extend the operation of substitution to contexts by setting

$$\Gamma[t/x] = x_1 : A_1[t/x], \ldots, x_n : A_n[t/x]$$

whenever no variable x_i occurs in t.

Unlike the case of simply-typed λ-calculus, it might happen that a context is not *well-formed*, in the sense that we do not expect it to make sense. For instance, if $\mathrm{Vec}\,n$ is the type of vectors of length n, the context

$$n : \mathrm{Nat}, l : \mathrm{Vec}\,n$$

which declares that n is a natural number and l is a vector of length n, is well-formed. However, the context

$$l : \mathrm{Vec}\,n, n : \mathrm{Nat}$$

is not well-formed: we begin by declaring that l is a vector of length n, without having declared what n should be before: the order in which variables are declared now really matters. Similarly, the context

$$n : \mathrm{Bool}, l : \mathrm{Vec}\, n$$

is not well-formed: the type $\mathrm{Vec}\, n$ only makes sense when n is a natural number, and not a boolean.

8.1.4 Definitional equality. In order to have a manageable type theory, we should identify some terms. In particular, we want to identify terms which are β-equivalent. For instance, suppose that we have a function f whose type is $\mathrm{Vec}(2 + 2) \to A$, taking a vector of length $2 + 2$ as argument and returning a term of type A, and a term t of type $\mathrm{Vec}(3+1)$. We expect to be able to apply f to t even though the types do not match precisely: the term t can be thought of as having the type $\mathrm{Vec}(2 + 2)$, because we all know that $2 + 2 = 3 + 1$. This means that in types, we consider terms up to some equivalence relation, called *convertibility* or *definitional equality*, which usually only consists in reduction.

Although we will formalize this definitional equality as an equivalence relation, we need some more properties on it: we need to be able to decide whether two terms are equivalent or not. In practice, this is performed by generalizing the method described in section 4.2.4 to test the β-convertibility of λ-terms: we orient the equivalence in a way giving rise to a convergent (i.e. terminating and confluent) relation, so that two terms are equivalent if and only if they have the same normal form. For instance, if our addition satisfies $(n+1)+m = n+(m+1)$ and $0 + m = m$, we orient those relations as $(n + 1) + m \rightsquigarrow n + (m + 1)$ and $0+m \rightsquigarrow m$. In order to know whether two expressions involving sums of natural numbers are equal, we can then apply those relations as much as possible, and compare the resulting expressions for equality. For instance,

$$2 + 2 \rightsquigarrow 1 + 3 \rightsquigarrow 0 + 4 \rightsquigarrow 4 \qquad \text{and} \qquad 1 + 3 \rightsquigarrow 0 + 4 \rightsquigarrow 4$$

and therefore the two terms are equivalent because they have the same normal form 4.

8.1.5 Sequents. In order to take all of this in account, we need to have three different forms of *judgments* in the sequent calculus:

- $\Gamma \vdash$ means that Γ is a well-formed context,

- $\Gamma \vdash t : A$ means that t has type A in the context Γ,

- $\Gamma \vdash t = u : A$ means that t and u are equal (i.e. convertible) terms of type A in the context Γ.

As usual, we will give rules which allow the derivation of those judgments through derivation trees. The derivation rules for all these three kinds of judgments mutually depend on each other, so that they all have to be defined at once.

As indicated above, there is no syntactic distinction between terms and types: both are expressions. The logic will however allow us to distinguish between the two. An expression A for which $\Gamma \vdash A : \mathrm{Type}$ is derivable for some context Γ is called a *type*. An expression t for which $\Gamma \vdash t : A$ is derivable for some context Γ and type A is called a *term*.

8.1.6 Rules for contexts. There are two rules for contexts:

$$\frac{}{\varnothing \vdash} \qquad\qquad \frac{\Gamma \vdash A : \mathrm{Type}}{\Gamma, x : A \vdash}$$

The first one states that the empty context $\varnothing$ is always well-formed. The second one states that if A is a well-formed type in a context Γ, then $\Gamma, x : A$ is a well-formed context. In the second rule, one would expect that we require that Γ is a well-formed context as a premise, as in

$$\frac{\Gamma \vdash \qquad \Gamma \vdash A : \mathrm{Type}}{\Gamma, x : A \vdash}$$

but we will see in section 8.1.11 that from the premise $\Gamma \vdash A : \mathrm{Type}$, we will actually be able to deduce that Γ is a well-formed context (and similar observations could be made on subsequent rules). As indicated above, the reason why we need to ensure that A is a well-formed type in the context Γ is to avoid considering a context such as

$$n : \mathrm{Bool}, l : \mathrm{Vec}\, n$$

as a well-formed context. Namely, the rules will not allow to derive

$$n : \mathrm{Bool} \vdash \mathrm{Vec}\, n : \mathrm{Type}$$

i.e. that $\mathrm{Vec}\, n$ is a well-formed type in a context were n is a boolean.

8.1.7 Rules for equality. We now give the rules for definitional equality. First, we have three rules ensuring that equality is an equivalence relation, by respectively imposing reflexivity, symmetry and transitivity:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A} \qquad \frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A} \qquad \frac{\Gamma \vdash t = u : A \qquad \Gamma \vdash u = v : A}{\Gamma \vdash t = v : A}$$

We will need that the definitional equality is not only an equivalence relation, but a congruence: rules expressing compatibility with type constructors will be added later on for each type constructor.

Finally, we add rules expressing the fact that a type can be substituted by an equal one in a typing derivation:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B : \mathrm{Type}}{\Gamma \vdash t : B} \qquad \frac{\Gamma \vdash t = u : A \qquad \Gamma \vdash A = B : \mathrm{Type}}{\Gamma \vdash t = u : B}$$

Example 8.1.7.1. The example of section 8.1.4, where a function f expecting an argument of type $\mathrm{Vec}(2+2)$ is applied to an argument l of type $\mathrm{Vec}(1+3)$, can be typed using these conversion rules as follows:

$$\frac{\dfrac{}{\ldots \vdash f : \mathrm{Vec}(2+2) \to A}\ (\mathrm{ax}) \qquad \dfrac{\dfrac{}{\ldots \vdash l : \mathrm{Vec}(1+3)}\ (\mathrm{ax}) \qquad \dfrac{\dfrac{\vdots}{\dfrac{\ldots \vdash 1+3 = 2+2 : \mathrm{Nat}}{\ldots \vdash \mathrm{Vec}(1+3) = \mathrm{Vec}(2+2)}}}{\ldots \vdash l : \mathrm{Vec}(2+2)}}{f : \mathrm{Vec}(2+2) \to A, l : \mathrm{Vec}(1+3) \vdash f\, l : A}}$$

8.1.8 Axiom rule. We now turn to rules allowing the typing of a term. The *axiom rule* is

$$\frac{\Gamma, x : A, \Gamma' \vdash}{\Gamma, x : A, \Gamma' \vdash x : A} \text{ (ax)}$$

with the following side conditions:

– $x \notin \text{dom}(\Gamma')$, and

– $\text{FV}(A) \cap \text{dom}(\Gamma') = \emptyset$.

We follow the convention that a variable always refers to the rightmost occurrence of the variable in a context. With this in mind, the side conditions avoid clearly wrong derivations such as

$$\frac{}{x : A, x : B \vdash x : A} \text{ (ax)} \qquad \frac{}{n : \text{Nat}, l : \text{Vec}\, n, n : \text{Bool} \vdash l : \text{Vec}\, n} \text{ (ax)}$$

Alternatively, we could use the convention that the variables declared in a context are always distinct, which we can always do because we consider terms up to α-conversion, although this is a bad habit because we do not want to spend our time performing α-conversions when implementing a proof assistant.

8.1.9 Terms and rules for type constructors. We now give the rules for Π-types, which are generalized function types. As for any type constructor in this type theory, we will need to have three constructions for expressions:

– a constructor for the type,

– a constructor for the terms of this type,

– an eliminator for the terms of this type,

together with six rules with the following purpose

– *formation*: construct a type with the type constructor,

– *introduction*: construct a term of the type,

– *elimination*: use a term of the type,

– *computation*: (β-)reduce a term of the type,

– *uniqueness*: express a uniqueness property of the constructed terms, which corresponds to an η-equivalence rule,

– *congruence*: express that definitional equality is compatible with the term constructors.

We insist here on this structure because it will be the same for all the subsequent type constructors that we are going to see in section 8.3. Let's see that in action for Π-types.

8.1.10 Rules for Π-types. The Π-types are dependent function types: they are like the plain old function types, except that the type of the result might depend on the argument. Such a type is written

$$\Pi(x : A).B$$

which corresponds to the Agda notation

$$(\text{x : A}) \to \text{B}$$

and should be read as the type of functions taking an argument x of type A and returning a value of type B. Here, the variable x might occur in the type B, i.e. the type B can *depend* on x. For instance, a function taking a natural number n as argument and returning a vector of length n will have the type

$$\Pi(n : \text{Nat}).\text{Vec}\, n$$

see section 6.4.7 for actual uses of such functions. In a Π-type as above, the variable x is bound in the type B, and we can rename bound variables. For instance, the previous type is α-equivalent to $\Pi(m : \text{Nat}).\text{Vec}\, m$. From a logical point of view, a type $\Pi(x : A).B$, can be read as a universal quantification

$$\forall x \in A.B$$

If we follow the lists given in section 8.1.9, the corresponding constructors for expressions are

– the constructor for types: Π,

– the constructor for terms: the λ-abstraction,

– the eliminator for terms: the application.

Finally, we can give the six required rules for Π-types.

Formation. The type formation rule is

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Pi(x : A).B : \text{Type}}(\Pi_{\text{F}})$$

and allows constructing a type $\Pi(x : A).B$ whenever A and B are well-formed types.

Introduction. The introduction rule is

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : \Pi(x : A).B}(\Pi_{\text{I}})$$

and states that a λ-abstraction $\lambda x^A.t$ is a function taking an argument x of type A and returning a term t of some type B: it should thus have the type $\Pi(x : A).B$.

Elimination. The elimination rule is

$$\frac{\Gamma \vdash t : \Pi(x : A).B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B[u/x]}(\Pi_{\mathrm{E}})$$

and states that if t is a function of type $\Pi(x : A).B$ and u is an argument of type A then we can apply t to u. Again, note that the type B can depend on x, so that the type of the result $t\,u$ should be B where x has been replaced by the actual value u of the argument.

Computation. The computation rule is

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x^A.t)\,u = t[u/x] : B}(\Pi_{\mathrm{C}})$$

this is precisely the β-reduction rule.

Uniqueness. The uniqueness rule is

$$\frac{\Gamma \vdash t : \Pi(x : A).B}{\Gamma \vdash t = \lambda x^A.t\,x : \Pi(x : A).B}(\Pi_{\mathrm{U}})$$

this is precisely the η-expansion rule.

Congruence. The three congruence rules are

$$\frac{\Gamma \vdash A = A' : \mathrm{Type} \qquad \Gamma, x : A \vdash B = B' : \mathrm{Type}}{\Gamma \vdash \Pi(x : A).B = \Pi(x : A').B' : \mathrm{Type}}$$

$$\frac{\Gamma \vdash A = A' : \mathrm{Type} \qquad \Gamma, x : A \vdash B : \mathrm{Type} \qquad \Gamma, x : A \vdash t = t' : B}{\Gamma \vdash \lambda x^A.t = \lambda x^{A'}.t' : \Pi(x : A).B}$$

and

$$\frac{\Gamma \vdash t = t' : \Pi(x : A).B \qquad \Gamma \vdash u = u' : A}{\Gamma \vdash t\,u = t'\,u' : B[t/x]}$$

They express the expected compatibility of equality with all the constructors for expressions: Π-types, λ-abstractions, and applications. We will generally omit the congruence rules in the following, but they should be formulated in a similar way for every constructor.

Example 8.1.10.1. The polymorphic identity function, which takes a type A and returns the identity function from A to A can be typed as follows:

$$\frac{\dfrac{\vdots}{\dfrac{A : \mathrm{Type}, x : A \vdash}{\dfrac{A : \mathrm{Type}, x : A \vdash x : A}{\dfrac{A : \mathrm{Type} \vdash \lambda x^A.x : \Pi(x : A).A}{\vdash \lambda A^{\mathrm{Type}}.\lambda x^A.x : \Pi(A : \mathrm{Type}).\Pi(x : A).A}(\Pi_{\mathrm{I}})}(\Pi_{\mathrm{I}})}(\mathrm{ax})}}{}$$

Arrow types. The traditional arrow type $A \to B$ can be recovered as the particular case of a Π-type $\Pi(x : A).B$ which is not dependent, meaning that x does not occur as a free variable in B. We thus write

$$A \to B \qquad = \qquad \Pi(\_ : A).B$$

where "$\_$" is a variable name which is supposed to never occur in any type; in particular, we always have $B[t/\_] = B$. It can be checked that all the rules give back the usual ones, up to notation. For instance, (Π_{E}) allows us to recover the elimination rule:

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}(\to_{\mathrm{E}})$$

8.1.11 Admissible rules. Many basic properties of the logical system can be expressed as the admissibility of some rules, some of which we now present. We concentrate on typing rules, i.e. judgments of the form $\Gamma \vdash t : A$, but similar admissible rules can usually be formulated for the two other kinds of judgments: well-formation of contexts ($\Gamma \vdash$) and convertibility ($\Gamma \vdash t = u : A$), details being left to the reader. The proofs are, as usual, performed by induction on the derivation of the judgment in the premise.

Before stating those, we first make the following simple, but useful, observation:

Lemma 8.1.11.1. For every derivable sequent $\Gamma \vdash t : A$, we have the inclusions $\mathrm{FV}(t) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{FV}(A) \subseteq \mathrm{dom}(\Gamma)$.

Proof. By induction on the derivation of the sequent. □

Basic checks. The rules ensure that only well-formed types and contexts can be manipulated at any point in a proof. This can be formulated as the admissibility of the following rules:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash} \qquad\qquad\qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash A : \mathrm{Type}}$$

$$\frac{\Gamma \vdash t = u : A}{\Gamma \vdash} \qquad \frac{\Gamma \vdash t = u : A}{\Gamma \vdash t : A} \qquad \frac{\Gamma \vdash t = u : A}{\Gamma \vdash u : A} \qquad \frac{\Gamma \vdash t = u : A}{\Gamma \vdash A : \mathrm{Type}}$$

To be honest the admissibility of those rules is "almost" true: this will be discussed in section 8.2.

Weakening rule. The following *weakening* rule is admissible, accounting for the fact that if some typing judgment holds in some context, it also holds with more hypothesis in the context.

$$\frac{\Gamma \vdash A : \mathrm{Type} \qquad \Gamma, \Gamma' \vdash t : B}{\Gamma, x : A, \Gamma' \vdash t : B}(\mathrm{wk})$$

with $x \notin \mathrm{FV}(\Gamma') \cup \mathrm{FV}(t) \cup \mathrm{FV}(B)$.

Exchange rule. The *exchange rule* states that we can swap two entries $x : A$ and $y : B$ in a context, provided that there is no dependency between them, i.e. B does not have x as free variable:

$$\frac{\Gamma \vdash B : \text{Type} \qquad \Gamma, x : A, y : B, \Delta \vdash t : C}{\Gamma, y : B, x : A, \Delta \vdash t : C}$$

Here, the hypothesis $\Gamma \vdash B : \text{Type}$ ensures that B does not depend on x by lemma 8.1.11.1.

Cut rule. The type theory has the cut elimination property, which corresponds to the admissibility of the following rule:

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A, \Delta \vdash u : B}{\Gamma, \Delta[t/x] \vdash u[t/x] : B[t/x]}(\text{cut})$$

see sections 2.3.3 and 4.1.8.

8.2 Universes

8.2.1 The type of Type**.** There is one missing thing in the type theory we have given up to now. Everything should have a type in the sequent we manipulate, but the constant Type does not, because there is no rule allowing us to do so. For instance, in order to type the polymorphic identity in example 8.1.10.1, we have to show that the context

$$A : \text{Type}, x : A \vdash x : A$$

is well-formed, which will at some point require showing

$$\vdash \text{Type} : \text{Type}$$

which we have no rule to derive for now.

There is an obvious candidate for the rule we are lacking: we are tempted to add the rule

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Type} : \text{Type}}$$

which is sometimes called the *type-in-type rule*. This rule was in fact present in the original Martin-Löf type system, but Girard showed that the resulting system was inconsistent [Gir72]. A variant of this proof is presented below.

8.2.2 Russell's paradox in type theory. We show the inconsistency with the above rule for Type by encoding, in Agda, Russell's paradox presented in section 5.3.1.

Encoding finite sets in OCaml. As a starter let's first see how to implement finite sets in OCaml. A finite set

$$A = \{a_1, \ldots, a_n\}$$

whose elements a_i belong to some fixed type `'a`, can be described by giving its elements: we can encode it as an array of elements. We thus define the type of sets of elements of `'a` as

```
type 'a finset = Finset of 'a array
```

which is a type, with one constructor, which takes an array of 'a as argument. This is thus essentially an array of 'a, and the usefulness of the constructor shall be explained below. It should be noted that this representation is not faithful: two arrays differing only by the order of their elements or repetitions of elements represent the same set. For instance, the arrays

$$[|3;1;2|] \qquad \text{and} \qquad [|2;1;3;1;2;2|]$$

both encode the set $\{1, 2, 3\}$.

We can code the function which determines whether an element x belongs to a set a, by looking whether the element occurs in the array:

```
let mem (x : 'a) (Finset a : 'a finset) =
  let ans = ref false in
  for i = 0 to Array.length a - 1 do
    if a.(i) = x then ans := true
  done;
  !ans
```

or, more elegantly, using the standard library, as

```
let mem (x : 'a) (Finset a : 'a finset) =
  Array.exists (fun y -> x = y) a
```

Similarly, inclusion of sets can be coded by

```
let included (Finset a : 'a finset) (b : 'a finset) =
  Array.for_all (fun x -> mem x b) a
```

Finally, the equality of two sets can be tested with

```
let eq (a : 'a finset) (b : 'a finset) =
  included a b && included b a
```

This is the right function to test equality of sets, which does not distinguish between two representations of the same set, and should always be used to compare sets, as opposed to the standard equality =.

In order to get closer to set theory, we shall now implement finite sets whose elements are themselves finite sets, i.e. we now consider the type

```
type finset = Finset of finset array
```

The previous functions are now mutually recursive because membership should be tested with respect to the suitable notion of equality:

```
let rec mem (x : 'a) (Finset a : finset) =
  Array.exists (fun y -> eq x y) a

and included (Finset a : finset) (b : finset) =
  Array.for_all (fun x -> mem x b) a

and eq (a : finset) (b : finset) =
  included a b && included b a
```

Encoding set theory in type theory. We can play the same game in type theory and define finite sets of elements in a type `A` in the same way. Instead of using arrays however, it is more natural to encode a finite set as a function of type

```
Fin n → A
```

for some natural number n: such a function f encodes the set

$$\{f(0), f(1), \dots, f(n-1)\}$$

(we recall that `Fin n` is the type whose elements are (isomorphic to) natural numbers from 0 to $n-1$). We can thus define

```
data finset (A : Set) : Set where
  Finset : {n : ℕ} → (Fin n → A) → finset A
```

In order to define "sets" of elements of `A`, instead of finite ones, we can allow indexing by any type instead of `Fin n`. Finally, we can encode sets (in the sense of type theory) as sets of sets. This suggests the following encoding of sets

```
data U : Set₁ where
  set : (I : Set) → (I → U) → U
```

which is due to Aczel [Acz78, Wer97]: a set consists of a type `I` of indices and a function which assigns a set to every element of `I`. In order to avoid confusion with the notation `Set` of Agda, we write `U` for the type of our sets.

   With this encoding the usual constructions can be performed. For instance, we can define the empty set:

```
∅ : U
∅ = set ⊥ (λ ())
```

the pairing of two sets:

```
⟨_,_⟩ : (A B : U) → U
⟨ A , B ⟩ = set Bool (λ {false → A ; true → B})
```

the product of two sets:

```
prod : (A B : U) → U
prod (set I f) (set J g) =
  set (I × J) (λ { (i , j) → ⟨ f i , g j ⟩ })
```

the equality of two sets (which implements the extensionality axiom):

```
_==_ : (A B : U) → Set
set I f == set J g =
  ((i : I) → Σ J (λ j → f i == g j)) ∧
  ((j : J) → Σ I (λ i → f i == g j))
```

the membership relation:

```
_∈_ : (A B : U) → Set
A ∈ set I f = Σ I (λ i → A == f i)
```

the union of sets (which implements the axiom of union):

```
⋃_ : (A : U) → U
⋃ set I f =
  set (Σ I (λ i → dom (f i))) (λ { (i , j) → F (f i) j })
```

where the domain is given by

```
dom : U → Set
dom (set I _) = I
```

and the function is given by

```
F : (A : U) → dom A → U
F (set _ f) = f
```

the von Neumann natural numbers (which implements the axiom of infinity):

```
vonN : ℕ → U
vonN ℕ.zero = ∅
vonN (ℕ.suc n) = ⋃ ⟨ vonN n , [ vonN n ] ⟩

Nat : U
Nat = set ℕ vonN
```

and so on.

The Russell paradox. Now, suppose that we accept this type-in-type rule which tells us that Type has type Type. This behavior can be achieved in Agda by using the flag

```
{-# OPTIONS --type-in-type #-}
```

at the beginning of the file. As before, we define sets as

```
data U : Set where
  set : (I : Set) → (I → U) → U
```

(the careful reader will notice that the type of U is now Set instead of Set_1), and we consider the following notion of membership

```
_∈_ : (A B : U) → Set
A ∈ set I f = Σ I (λ i → f i ≡ A)
```

This function is "wrong" because equality is tested here using propositional equality instead of the proper equality == between sets, but it will be enough for the purpose of implementing paradoxes and give rise to shorter code. We declare a set to be *regular* if it does not contain itself, which can be defined by

```
regular : U → Set
regular A = ¬ (A ∈ A)
```

and consider Russell's paradoxical set R of all sets which do not contain themselves, see section 5.3.1:

```
R : U
R = set (Σ U (λ A → regular A)) proj₁
```

This set can be shown to be both regular

```
R-nonreg : ¬ (regular R)
R-nonreg reg = reg ((R , reg) , refl)
```

and non-regular

```
R-reg : regular R
R-reg ((A , reg) , p) = subst regular p reg ((A , reg) , p)
```

from which we can deduce the inconsistency of the system:

```
absurd : ⊥
absurd = R-nonreg R-reg
```

8.2.3 Girard's paradox. It is always good to have a handful of paradoxes at hand in order to test a proof assistant: depending on the logic, one or the other might be easier to encode. For instance, the above formalization crucially depends on the fact that we have inductive types, which is not the case of all proof assistants. As another example, we shall present Girard's original paradox [Gir72], which is based on the following set-theoretic paradox which shows that there is no ordinal of all ordinals.

The Burali-Forti paradox. A *well-ordered set* is traditionally defined as a set A equipped with a total order $\leqslant$ which is well-founded, i.e. there is no infinite strictly decreasing sequence of elements, see section 6.8.6 and appendix A.3. Alternatively – and this is better suited to formalization – a well-ordered set can be defined as a set A equipped with a relation $<$ which is

   – transitive,

   – well-founded, and

   – extensional.

By *extensional*, we mean here that, given $y, z \in A$, if $x < y$ is equivalent to $x < z$ for every $x \in A$, then $y = z$:

$$(\forall x \in A.x < y \Leftrightarrow x < z) \Rightarrow y = z$$

Two well-ordered sets are *isomorphic* when they are in bijection with order-preserving functions. An *ordinal* is the isomorphism class of a well-ordered set. An *embedding* of an ordinal A into an ordinal B is an increasing function f from A to B, i.e. such that $x < y$ implies $f(x) < f(y)$ for every $x, y \in A$. Such an embedding is *bounded* when there exists an element $b \in B$ such that $f(x) < b$ for every $x \in A$. We define a relation $<$ on ordinals by setting $A < B$ whenever there exists a bounded embedding of A into B. This relation can be shown to be transitive, well-founded and extensional.

The *Burali-Forti paradox* [BF97] shows that the ordinal numbers do not form a set: they are too big to be so, in the same sense that the collection of all sets is too big to itself be a set. Namely, suppose that there is a set Ω of all ordinals. By the above, when equipped with the relation $<$, this would induce an ordinal that we still write Ω. It can be shown that for every ordinal A, we have $A < \Omega$. In particular, we have $\Omega < \Omega$, which is in contradiction with the hypothesis that $<$ should be well-founded. Details to follow.

Formalizing Girard's paradox. The Girard paradox [Gir72] is an implementation of the above paradox in Martin-Löf type theory with the type-in-type rule. A nice account of this paradox can also be found in the introduction of [ML98]. We present here a formalization of it.

The notion of *ordinal* can be formalized in Agda by a record

```
record Ord : Set where
  field
    car   : Set
    rel   : Rel car
    trans : Transitive rel
    wf    : WellFounded rel
```

It is a 4-uple consisting of a *carrier* type car and a relation rel on it (see section 6.5.9), together with a proof that this relation is transitive and well-founded. Here, the predicate of being transitive for a relation is defined by

```
Transitive : {A : Set} → Rel A → Set
Transitive {A} R = {x y z : A} → R x y → R y z → R x z
```

and well-foundedness is detailed in section 6.8.6. The above definition actually formalizes a generalization of the notion of ordinal: in order to define traditional ones, we should also impose that they are extensional, i.e. a proof of Extensional rel, where the extensionality predicate is

```
Extensional : {A : Set} → Rel A → Set
Extensional {A} R =
  (y y' : A) → ((x : A) → R x y ↔ R x y') → y ≡ y'
```

but this will play no role in our proof so that we omit it for simplicity (we refer the reader to [Uni13, Section 10.3] for a detailed formalization of ordinals). Given an ordinal A, we use the more readable notation ‖ A ‖ for its carrier:

```
‖_‖ : Ord → Set
‖ A ‖ = car A
```

Since ordinals are well-founded, we can use the following induction principle in order to reason about those:

```
Ord-rec : (A : Ord) → (P : ‖ A ‖ → Set) →
          ((x : ‖ A ‖) → ((y : ‖ A ‖) → rel A y x → P y) → P x) →
          (x : ‖ A ‖) → P x
Ord-rec A = wfRec (wf A)
```

An *embedding* of an ordinal to the other consists of a function between the underlying carriers together with a proof that it is increasing, and we write Emb A B for the type of embeddings of A into B:

```
record Emb (A B : Ord) : Set where
  field
    fun : ‖ A ‖ → ‖ B ‖
    inc : ∀ {x y} → rel A x y → rel B (fun x) (fun y)
```

Such an embedding is *bounded* by b in B when the image of every element of A is below b and we write BEmb A B b for the type of embeddings of A into B which are bounded by b:

```
record BEmb (A B : Ord) (b : ∥ B ∥) : Set where
  field
    emb : Emb A B
    bnd : (x : ∥ A ∥) → rel B (fun emb x) b
```

Based on this, we can define the relation < on ordinals, such that A < B whenever there is a bounded embedding of A into B.

```
_<_ : Ord → Ord → Set
A < B = Σ ∥ B ∥ (λ b → BEmb A B b)
```

This relation is easily shown to be transitive

```
<-trans : Transitive _<_
<-trans (y , f) (z , g) = (fun (emb g) y) , (comp f g)
```

and well-founded with some more work:

```
<-wf : WellFounded _<_
<-wf A = acc lem
  where
  lem : (B : Ord) → B < A → Acc _<_ B
  lem B (a , f) = Ord-rec A P' lem' a B f
    where
    P' : ∥ A ∥ → Set
    P' a = (B : Ord) → BEmb B A a → Acc _<_ B
    lem' : (a : ∥ A ∥) → ((a' : ∥ A ∥) → rel A a' a → P' a') → P' a
    lem' a ind B f =
      acc (λ { C (b , g) →
                ind (fun (emb f) b) (bnd f b) C (comp g f) })
```

In words: showing that this relation is well-founded amounts to showing that every ordinal A is accessible, which by definition of accessibility amounts to showing that every ordinal B with $B < A$ is accessible. By definition of the relation $<$ on ordinals, this amounts to showing that for every embedding $f : B \to A$ bounded by $a \in A$, we have that B is accessible. This last property is written $P'(a)$ and shown by induction on $a \in A$ (with respect to the order $<_A$ on the elements of the ordinal A, which is well-founded). Supposing that the property $P'(a')$ hold for every $a' \in A$ with $a' <_A a$, we have to show $P(a)$, that is, given an embedding $f : B \to A$ bounded by a, that B is accessible. By definition of accessibility, this amounts to showing that C is accessible for every $C < B$. Given such an ordinal C, the fact that $C < B$ means that there exists an embedding $g : C \to B$ which is bounded by $b \in B$. By composing f and g, we therefore have an embedding $f \circ g : C \to A$ which is bounded by $f(b)$ and we conclude that C is accessible by applying $P'(f(b))$. In the above proof, the composition of the embedding is handled by the function

```
comp : {A B C : Ord} {b : ∥ B ∥} {c : ∥ C ∥}
       (f : BEmb A B b) (g : BEmb B C c) →
       BEmb A C (fun (emb g) b)
```

whose proof is left to the reader. If we suppose that we have the type-in-type rule with

```
{-# OPTIONS --type-in-type #-}
```

we can then define the ordinal Ω of all ordinals:

```
Ω : Ord
car   Ω = Ord
rel   Ω = _<_
trans Ω = <-trans
wf    Ω = <-wf
```

We can now show that Ω is the maximal element of ordinals:

```
A<Ω : {A : Ord} → A < Ω
A<Ω {A} = A , record {
  emb = record {
    fun = λ x → A ↓ x ;
    inc = λ {x} {y} x<y → ↓-inc A x<y } ;
  bnd = λ x → x , snd (↓-< A x) }
```

Above, given an ordinal A, we show that we have $A < \Omega$: we need to construct a bounded embedding $f : A \to \Omega$. Here, we take the function which takes an element $x \in A$ to the ordinal $A \downarrow x$ defined as the restriction of A to the elements smaller than x, i.e.

```
_↓_ : (A : Ord) → ‖ A ‖ → Ord
car   (A ↓ a) = Σ ‖ A ‖ (λ x → rel A x a)
rel   (A ↓ a) x y = rel A (fst x) (fst y)
trans (A ↓ a) x<y y<z = trans A x<y y<z
wf    (A ↓ a) = ↓wf
```

The proof of well-foundedness ↓wf, deduced from the well-foundedness of A, is left to the reader. The proofs that the embedding is increasing

```
↓-inc : (A : Ord) {a b : ‖ A ‖} → rel A a b → A ↓ a < A ↓ b
```

and bounded

```
↓-< : (A : Ord) (a : ‖ A ‖) → (A ↓ a) < A
```

are also left to the reader. As a particular case of the above lemma, we have that $\Omega < \Omega$:

```
Ω<Ω : Ω < Ω
Ω<Ω = A<Ω
```

which contradicts the fact that the relation < is well-founded. Namely, any relation R with $x\,R\,x$ will have an infinite decreasing sequence: namely $x\,R\,x\,R\,x\,R\ldots$ Constructively, this is of course shown by induction:

```
wf-irrefl : {A : Set} (R : Rel A) → WellFounded R →
            (x : A) → R x x → ⊥
wf-irrefl R wf x =
  wfRec wf (λ x → R x x → ⊥) (λ y ind Ryy → ind y Ryy Ryy) x
```

From which we see that accepting Ω as an ordinal leads to the system being inconsistent:

```
absurd : ⊥
absurd = wf-irrefl _<_ <-wf Ω Ω<Ω
```

Variants and other paradoxes. The Girard paradox is analyzed by Coquand in [Coq86] and simplified by Hurkens [Hur95]. Some other paradoxes have also been produced by Coquand [Coq92a, Coq95]. For instance, one is, as translated by Abel [Abe17]:

```
{-# OPTIONS --type-in-type #-}

data U : Set where
  c : ({A : Set} → A → A) → U

empty : {A : Set} → U → A
empty (c f) = empty (f (c (λ z → z)))

absurd : {A : Set} → A
absurd = empty (c (λ z → z))
```

8.2.4 The hierarchy of universes. How should we fix this? If we think of the situation we already faced when considering naive set theory, the explanation was that the collection of all sets was "too big" to be a set. Similarly, we think of Type as being "too big" to be a type. However, we still need to give it a type, and the natural next move is to introduce a new constructor, say TYPE, which is the type of "big types", together with the rule

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathrm{Type} : \mathrm{TYPE}}$$

stating that Type is a big type. However, we now need to give a type to TYPE, which forces us to introduce a type of "very big types" and so on.

In the end, we introduce a hierarchy of types Type_i indexed by natural numbers $i \in \mathbb{N}$, together with the rule

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathrm{Type}_i : \mathrm{Type}_{i+1}}$$

for every $i \in \mathbb{N}$. The type Type is simply a notation for Type_0, Type_1 is the type of "big types", Type_2 is the type of "very big types", Type_3 is the type of "very very big types", and so on:

$$\mathrm{Type}_0 : \mathrm{Type}_1 : \mathrm{Type}_2 : \mathrm{Type}_3 : \dots$$

The types Type_i are called *universes* and i is called the *level* of the universe Type_i. In order to make the theory more manageable, we also add a *cumulativity rule*

$$\frac{\Gamma \vdash A : \mathrm{Type}_i}{\Gamma \vdash A : \mathrm{Type}_{i+1}}$$

which states that a "small" type can always be seen as a "bigger" type. This allows us to see a type in a given universe as a type in a universe of higher level, so that all constructions can be cast in to higher levels if necessary and we do not have to precisely take care of the levels. Finally, we change all the

type formation rules by adding levels to occurrences of Type. For instance, the formation rule for Π-types becomes

$$\frac{\Gamma \vdash A : \text{Type}_i \qquad \Gamma, x : A \vdash B : \text{Type}_i}{\Gamma \vdash \Pi(x : A).B : \text{Type}_i} \ (\Pi_\text{F})$$

In the following, except in this section, we will not be precise on the universe levels and still allow us to use the type Type. However, it can be checked that all the subsequent constructions can be adapted as above in order to properly take levels in account.

Universes in Agda. In Agda, `Set` is a notation for Type_0, `Set`$_1$ is a notation for Type_1 and so on. For instance, we can define the type of predicates on a type A as

```
Predicate : (A : Set) → Set₁
Predicate A = A → Set
```

However, if we try to define `Predicate` as being of type `(A : Set) → Set`, Agda will complain that we are trying to fit a type in Type_1 into Type_0 by issuing the following error message:

```
Set₁ != Set
when checking that the expression A → Set has type Set
```

Cumulative universes. Systems like Coq have the cumulativity rule built-in, but systems such as Agda chose not to, mostly for technical reasons. Since we don't have it, the type formation rules now have to allow constructors to have different levels, and for instance the formation rule for Π-types has to be changed to

$$\frac{\Gamma \vdash A : \text{Type}_i \qquad \Gamma, x : A \vdash B : \text{Type}_j}{\Gamma \vdash \Pi(x : A).B : \text{Type}_{\max(i,j)}} \ (\Pi_\text{F})$$

We thus need three operations on levels i:

- we need to have a level 0,

- for every level i we need to have a successor level $i + 1$ (in order to type Type_i), and

- we need to be able to compute the maximum of two levels.

This why in Agda levels are defined in the module `Level` by

```
postulate
  Level : Set
  lzero : Level
  lsuc  : (i : Level) → Level
  _⊔_   : (i j : Level) → Level
```

Universe polymorphism. Up to now, we have been defining equality as

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

This means that we can use it to compare the elements of a small type (for instance, we can use it to compare natural numbers), but we cannot use it to compare elements of a large type, typically types. For instance, suppose that we want to show that the type $\mathbb{N}$ is different from the empty type $\bot$, i.e. we want to prove:

```
NB : ¬ (ℕ ≡ ⊥)
```

If we try to prove this with the above definition for equality, Agda complains that

```
Set₁ != Set when checking that the expression ℕ has type Set
```

This is because we are trying to compare the types $\mathbb{N}$ and $\bot$, which are of type `Set`, which is itself of type $\mathtt{Set}_1$, whereas equality is defined on elements of a type `A` whose type is `Set`. To overcome this, we could define another equality $\equiv_1$ which allows for comparing types:

```
data _≡₁_ {A : Set₁} (x : A) : A → Set where
  refl : x ≡₁ x
```

But this is quite unsatisfactory. Apart from the subscript "$_1$", this definition is essentially the same as the one for $\equiv$, and we have to prove again for this notion of equality all the properties we have already proved for $\equiv$, by copying the proofs and inserting "$_1$" from time to time, which means lots of duplication of code. Moreover, we would have to do this once again if we want to compare elements of a type whose type is $\mathtt{Set}_2$, $\mathtt{Set}_3$, and so on.

In order to solve this problem, Agda allows for defining functions on type Type_i for every level i: this is called *universe polymorphism*. This means that we can define functions which can take universe levels as arguments. For instance, we can define equality as

```
data _≡_ {i : Level} {A : Set i} (x : A) : A → Set where
  refl : x ≡ x
```

As you can observe, the Agda notation for Type_i is `Set i`. This definition depends on a universe level `i` which is implicit and thus automatically inferred, and thus allows for comparing elements of types of any level. The actual definition of equality in the standard library is this one and we can now finish our example with

```
NB : ¬ (ℕ ≡ ⊥)
NB p with coe p 0
NB p | ()
```

Lifting. As another application of universe polymorphism, we can derive in Agda the cumulativity of universes: we can construct a function `Lift` which takes a element of Type_i and casts it as an element of Type_j with $j > i$, called the *lifting* of the type. Since we do not have access to the order on levels, but

can compute the maximum $\sqcup$ of two levels, we actually rather give it the type $\mathrm{Type}_i \to \mathrm{Type}_{i \sqcup j}$, which also ensures that the returned level is greater than the one given as input. The definition is performed based on the observation that the lifted type should have the "same" elements as the original one, which can be expressed by the following inductive type:

```
data Lift {i} j (A : Set i) : Set (i ⊔ j) where
  lift : A → Lift j A
```

8.3 More type constructors

In this section, we give the rules in order to add many of the usual type constructors in dependent type theory. All those will be subsumed by inductive types introduced in section 8.4: as in Agda, these constructions can be implemented as particular inductive types.

8.3.1 Empty type. For the empty type, or falsity, we add the following two constructions to expressions

$$e ::= \ldots \mid \bot \mid \mathrm{bot}(e, x \mapsto e')$$

The type $\bot$ is the type for falsity, which is empty, and the construction

$$\mathrm{bot}(t, x \mapsto A)$$

eliminates a proof t of $\bot$ in order to construct an element of type A (which might depend on t via the variable x). The arrow $\mapsto$ is only a formal notation here, and does not mean a function: bot is a formal constructor which takes as argument an expression t, a variable x and an expression A. However, the variable x is bound in A, and could be renamed to any other variable name. In Agda, it would correspond to the operation which matches a proof of $\bot$ in order to produce an A, i.e. something like

```
bot : (x : ⊥) → A x
elim ()
```

The rules are as follows.

Formation. $\bot$ is a valid type in any valid context:

$$\frac{\Gamma \vdash}{\Gamma \vdash \bot : \mathrm{Type}} \ (\bot_{\mathrm{F}})$$

Introduction. There is no introduction rule, because we do not expect that there is a way to prove falsity.

Elimination. Elimination allows proving anything from falsity:

$$\frac{\Gamma \vdash t : \bot \qquad \Gamma, x : \bot \vdash A : \mathrm{Type}}{\Gamma \vdash \mathrm{bot}(t, x \mapsto A) : A[t/x]} \ (\top_{\mathrm{E}})$$

Computation. No rule.

Uniqueness. No rule.

8.3.2 Unit type. For the unit type, or truth, we add the following constructions to expressions:

$$e ::= \dots \mid \top \mid \star \mid \text{top}(e, x \mapsto e', e'')$$

where $\top$ is the type for truth, $\star$ is the constructor for truth and

$$\text{top}(t, x \mapsto A, u)$$

eliminates a proof t of $\top$ in order to construct a proof u of A.

Formation.

$$\frac{\Gamma \vdash}{\Gamma \vdash \top : \text{Type}} \ (\top_{\text{F}})$$

Introduction.

$$\frac{\Gamma \vdash}{\Gamma \vdash \star : \top} \ (\top_{\text{I}})$$

Elimination.

$$\frac{\Gamma \vdash t : \top \qquad \Gamma, x : \top \vdash A : \text{Type} \qquad \Gamma \vdash u : A[\star/x]}{\Gamma \vdash \text{top}(t, x \mapsto A, u) : A[t/x]} \ (\top_{\text{E}})$$

Computation.

$$\frac{\Gamma, x : \top \vdash A : \text{Type} \qquad \Gamma \vdash u : A[\star/x]}{\Gamma \vdash \text{top}(\star, x \mapsto A, u) = u : A[\star/x]} \ (\top_{\text{C}})$$

Uniqueness.

$$\frac{\Gamma \vdash t : \top}{\Gamma \vdash t = \star : \top} \ (\top_{\text{U}})$$

In OCaml. The type $\top$ corresponds to `unit`, the constructor $\star$ to `()`, the eliminator $\text{top}(t, x \mapsto A, u)$ to

```
match t with
| () -> u
```

the computation rule says that

```
match () with
| () -> u
```

evaluates to u, and uniqueness says that `()` is the only value of type `unit`.

8.3.3 Products. For the product, or conjunction, of two types, we add the
following constructions to expressions:

$$e ::= \dots \mid e \times e' \mid \langle e, e' \rangle \mid \operatorname{unpair}(e, x \mapsto e', \langle y, z \rangle \mapsto e'')$$

The type $A \times B$ is the product of A and B (it is sometimes also written $A \wedge B$).
The term $\langle t, u \rangle$ is the pair of two terms t and u and

$$\operatorname{unpair}(t, z \mapsto A, \langle x, y \rangle \mapsto u)$$

eliminates a pair t, extracting its components x and y, in order to construct a
proof u whose type is A which might depend on t as z.

Formation.

$$\frac{\Gamma \vdash A : \operatorname{Type} \qquad \Gamma \vdash B : \operatorname{Type}}{\Gamma \vdash A \times B : \operatorname{Type}} \ (\times_{\mathrm{F}})$$

Introduction.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} \ (\times_{\mathrm{I}})$$

Elimination.

$$\frac{\Gamma \vdash t : A \times B \qquad\qquad\qquad}{\dfrac{\Gamma, z : A \times B \vdash C : \operatorname{Type} \qquad \Gamma, x : A, y : B \vdash u : C[\langle x, y \rangle / z]}{\Gamma \vdash \operatorname{unpair}(t, z \mapsto C, \langle x, y \rangle \mapsto u) : C[t/z]}} \ (\times_{\mathrm{E}})$$

Computation.

$$\frac{\Gamma \vdash t : A \qquad\qquad\qquad\qquad}{\dfrac{\Gamma \vdash u : B \qquad \Gamma, z : A \times B \vdash C : \operatorname{Type} \qquad \Gamma, x : A, y : B \vdash v : C[\langle x, y \rangle / z]}{\Gamma \vdash \operatorname{unpair}(\langle t, u \rangle, z \mapsto C, \langle x, y \rangle \mapsto v) = v[t/x, u/y] : C[\langle t, u \rangle / z]}} \ (\times_{\mathrm{C}})$$

Uniqueness.

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \operatorname{unpair}(t, z \mapsto A \times B, \langle x, y \rangle \mapsto \langle x, y \rangle) = t : A \times B} \ (\times_{\mathrm{U}})$$

In OCaml. The type $A \times B$ corresponds to `a * b`, the term $\langle t, u \rangle$ to the pair
(`t` , `u`), the eliminator $\operatorname{unpair}(t, z \mapsto A, \langle x, y \rangle \mapsto u)$ to

```
match t with
| (x , y) -> u
```

the computation rule says that

```
match (t , u) with
| (x , y) -> v x y
```

evaluates to `v t u`, and the uniqueness rule says that

```
match t with
| (x , y) -> (x , y)
```

is the same as `t`.

8.3.4 Dependent sums. Dependent sums, or Σ-*types*, are a generalization the previous notion of product, where the type of the second component might depend on the term of the first component. Such a type is written

$$\Sigma(x : A).B$$

and the elements of this type are the pairs (t, u) consisting of a term t of type A and a term u of type $B[t/x]$. From a logical point of view, this corresponds to an existential quantification

$$\exists x \in A.B$$

Namely, a proof of such a proposition consists of a term t in A together with a proof that $B(t)$ is satisfied. Formally, we add the following constructions to expressions:

$$e ::= \dots \mid \Sigma(x : A).B \mid \langle e, e' \rangle \mid \mathrm{unpair}(e, x \mapsto e', \langle y, z \rangle \mapsto e'')$$

Formation.

$$\frac{\Gamma \vdash A : \mathrm{Type} \qquad \Gamma, x : A \vdash B : \mathrm{Type}}{\Gamma \vdash \Sigma(x : A).B : \mathrm{Type}} \ (\Sigma_{\mathrm{F}})$$

Introduction.

$$\frac{\Gamma, x : A \vdash B : \mathrm{Type} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/x]}{\Gamma \vdash \langle t, u \rangle : \Sigma(x : A).B} \ (\Sigma_{\mathrm{I}})$$

Elimination.

$$\frac{\Gamma \vdash t : \Sigma(x : A).B}{\Gamma, z : \Sigma(x : A).B \vdash C : \mathrm{Type} \qquad \Gamma, x : A, y : B \vdash u : C[\langle x, y \rangle / z]}{\Gamma \vdash \mathrm{unpair}(t, z \mapsto C, \langle x, y \rangle \mapsto u) : C[t/z]} \ (\Pi_{\mathrm{E}})$$

Computation.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/x]}{\Gamma, z : \Sigma(x : A).B \vdash C : \mathrm{Type} \qquad \Gamma, x : A, y : B \vdash v : C[\langle x, y \rangle / z]}{\Gamma \vdash \mathrm{unpair}(\langle t, u \rangle, z \mapsto C, \langle x, y \rangle \mapsto v) = v[t/x, u/y] : C[\langle t, u \rangle / z]} \ (\Pi_{\mathrm{C}})$$

Uniqueness.

$$\frac{\Gamma \vdash t : \Sigma(x : A).B}{\Gamma \vdash \mathrm{unpair}(t, z \mapsto \Sigma(x : A).B, \langle x, y \rangle \mapsto \langle x, y \rangle) = t : \Sigma(x : A).B} \ (\Pi_{\mathrm{U}})$$

Pairs. A pair $A \times B$ is a particular case of a Σ-type $\Sigma(x : A).B$ which is not dependent, i.e. $x \notin \mathrm{FV}(B)$. In other words, by setting

$$A \times B \qquad = \qquad \Sigma(\_ : A).B$$

where $\_$ is a variable which never occurs in B, we recover the rules previously given for products.

It might be puzzling at first that a product would correspond to a sum, but one should recall that this is actually already the case for natural numbers

$$m \times n = \sum_{i=0}^{m-1} n$$

Here, a dependent sum would rather correspond to summing a finite family $(n_i)_{0 \leqslant i < m}$ of natural numbers:

$$\sum_{i=0}^{m-1} n_i$$

and a product is a particular case of this where the family is constant (i.e. $n_i = n$ for every index i).

8.3.5 Coproducts. For coproducts, we add the following constructions to expressions:

$$e ::= \dots \mid e + e' \mid \iota_{\mathrm{l}}^e(e') \mid \iota_{\mathrm{r}}^e(e') \mid \mathrm{case}(x, e \mapsto y, e' \mapsto z, e'' \mapsto e''')$$

The type $A + B$ is the coproduct of A and B, which logically corresponds to their disjunction. The elements of this type are either a term t of A, written $\iota_{\mathrm{l}}^B(t)$, or a term u of B, written $\iota_{\mathrm{r}}^A(u)$, and the eliminator $\mathrm{case}(t, z \mapsto C, x \mapsto u, y \mapsto v)$ eliminates t to construct a term of type C (which might depend on t as x) by considering whether it is of the first or the second form, in which case u or v is returned.

Formation.

$$\frac{\Gamma \vdash A : \mathrm{Type} \qquad \Gamma \vdash B : \mathrm{Type}}{\Gamma \vdash A + B : \mathrm{Type}} \; (+_{\mathrm{F}})$$

Introduction.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : \mathrm{Type}}{\Gamma \vdash \iota_{\mathrm{l}}^B(t) : A + B} \; (+_{\mathrm{I}}^{\mathrm{l}}) \qquad\qquad \frac{\Gamma \vdash A : \mathrm{Type} \qquad \Gamma \vdash t : B}{\Gamma \vdash \iota_{\mathrm{r}}^A(t) : A + B} \; (+_{\mathrm{I}}^{\mathrm{r}})$$

Elimination.

$$\frac{\begin{array}{c} \Gamma \vdash t : A + B \qquad \Gamma, z : A + B \vdash C : \mathrm{Type} \\ \Gamma, x : A \vdash u : C[\iota_{\mathrm{l}}^B(x)/z] \qquad \Gamma, y : B \vdash v : C[\iota_{\mathrm{r}}^A(y)/z] \end{array}}{\Gamma \vdash \mathrm{case}(t, z \mapsto C, x \mapsto u, y \mapsto v) : C[t/z]} \; (+_{\mathrm{E}})$$

Computation.

$$\frac{\begin{array}{c} \Gamma \vdash t : A \qquad \Gamma \vdash B : \mathrm{Type} \qquad \Gamma, z : A \vdash C : \mathrm{Type} \\ \Gamma, x : A \vdash u : C[\iota_{\mathrm{l}}^B(x)/z] \qquad \Gamma, y : B \vdash v : C[\iota_{\mathrm{r}}^A(y)/z] \end{array}}{\Gamma \vdash \mathrm{case}(\iota_{\mathrm{l}}^B(t), z \mapsto C, x \mapsto u, y \mapsto v) = u[t/x] : C[\iota_{\mathrm{l}}^B(t)/z]} \; (+_{\mathrm{C}}^{\mathrm{l}})$$

$$\frac{\begin{array}{c} \Gamma \vdash A : \mathrm{Type} \qquad \Gamma \vdash t : B \qquad \Gamma, z : A \vdash C : \mathrm{Type} \\ \Gamma, x : A \vdash u : C[\iota_{\mathrm{l}}^B(x)/z] \qquad \Gamma, y : B \vdash v : C[\iota_{\mathrm{r}}^A(y)/z] \end{array}}{\Gamma \vdash \mathrm{case}(\iota_{\mathrm{r}}^A(t), z \mapsto C, x \mapsto u, y \mapsto v) = v[t/x] : C[\iota_{\mathrm{r}}^A(t)/z]} \; (+_{\mathrm{C}}^{\mathrm{r}})$$

Uniqueness.

$$\frac{\Gamma \vdash t : A + B}{\Gamma \vdash \mathrm{case}(t, z \mapsto A + B, x \mapsto \iota_\mathrm{l}^B(x), y \mapsto \iota_\mathrm{r}^A(y)) = t : A + B} \ (+_\mathrm{U})$$

In OCaml. The type $A + B$ corresponds to an inductive type of the form

```
type ('a , 'b) coprod =
  | Left  of 'a
  | Right of 'b
```

$\iota_\mathrm{l}^B(t)$ to `Left t`, $\iota_\mathrm{r}^A(t)$ to `Right t` and the eliminator

$$\mathrm{case}(t, z \mapsto C, x \mapsto u, y \mapsto v)$$

to

```
match t with
| Left  x -> u
| Right y -> v
```

The left computation rule says that

```
match Left t with
| Left  x -> u x
| Right y -> v y
```

reduces to `u x` (and similarly for the right one) and the uniqueness rule says that

```
match t with
| Left  x -> Left x
| Right y -> Right y
```

is the same as `t`.

In Agda. The standard notation for $+$ is ⊎ and the notations for ι_l and ι_r are respectively inj_1 and inj_2, see section 6.5.6.

8.3.6 Booleans. For booleans, we add the following constructions to expressions:

$$e ::= \ldots \mid \mathrm{Bool} \mid 1 \mid 0 \mid \mathrm{ite}(e, x \mapsto e', e'', e''')$$

where Bool is the type of booleans, 1 and 0 are true and false respectively and $\mathrm{ite}(t, x \mapsto A, u, v)$ is conditional which returns u or v depending on whether t is true or false.

Formation.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathrm{Bool} : \mathrm{Type}} \ (\mathrm{Bool_F})$$

Introduction.

$$\frac{\Gamma \vdash}{\Gamma \vdash 1 : \mathrm{Bool}} \ (\mathrm{Bool_I^1}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash 0 : \mathrm{Bool}} \ (\mathrm{Bool_I^0})$$

Elimination.

$$\frac{\Gamma \vdash t : \mathrm{Bool} \qquad \Gamma, x : \mathrm{Bool} \vdash A : \mathrm{Type} \qquad \Gamma \vdash u : A[1/x] \qquad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \mathrm{ite}(t, x \mapsto A, u, v) : A[t/x]} \ (\mathrm{Bool_E})$$

Computation.

$$\frac{\Gamma, x : \mathrm{Bool} \vdash A : \mathrm{Type} \qquad \Gamma \vdash u : A[1/x] \qquad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \mathrm{ite}(1, x \mapsto A, u, v) = v : A[1/x]} \ (\mathrm{Bool_C^1})$$

$$\frac{\Gamma, x : \mathrm{Bool} \vdash A : \mathrm{Type} \qquad \Gamma \vdash u : A[1/x] \qquad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \mathrm{ite}(0, x \mapsto A, u, v) = u : A[0/x]} \ (\mathrm{Bool_C^0})$$

Uniqueness.

$$\frac{\Gamma \vdash t : \mathrm{Bool}}{\Gamma \vdash \mathrm{ite}(t, x \mapsto \mathrm{Bool}, 1, 0) = t : \mathrm{Bool}} \ (\mathrm{Bool_U})$$

In OCaml. Bool corresponds to the type `bool`, 1 and 0 correspond to `true` and `false` respectively and the eliminator $\mathrm{ite}(x, A \mapsto u, v, t)$ corresponds to

```
if t then u else v
```

The computation rule says that

```
if true then u else v
```

reduces to `u` and that

```
if false then u else v
```

reduces to `v`, and the uniqueness rule says that

```
if t then true else false
```

is the same as `t`.

8.3.7 Natural numbers. For natural numbers, we add the following constructions to expressions:

$$e ::= \ldots \mid \mathrm{Nat} \mid \mathrm{Z} \mid \mathrm{S}(e) \mid \mathrm{rec}(e, x \mapsto e', e'', yz \mapsto e''')$$

where Nat is the type of natural numbers, Z is zero, $\mathrm{S}(t)$ is the successor of t and $\mathrm{rec}(z, A \mapsto u, x, yv \mapsto t)$ is the induction principle on t: u is the base case and t is the inductive case.

Formation.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathrm{Nat} : \mathrm{Type}} \ (\mathrm{Nat_F})$$

Introduction.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathrm{Z} : \mathrm{Nat}} \ (\mathrm{Nat_I^Z}) \qquad\qquad \frac{\Gamma \vdash t : \mathrm{Nat}}{\Gamma \vdash \mathrm{S}(t) : \mathrm{Nat}} \ (\mathrm{Nat_I^S})$$

Elimination.

$$\frac{\Gamma \vdash t : \mathrm{Nat} \qquad \Gamma, x : \mathrm{Nat} \vdash A : \mathrm{Type}}{\Gamma \vdash u : A[\mathrm{Z}/x] \qquad \Gamma, x : \mathrm{Nat}, y : A \vdash v : A[\mathrm{S}(x)/x]}{\Gamma \vdash \mathrm{rec}(t, x \mapsto A, u, xy \mapsto v) : A[t/x]} \ (\mathrm{Nat_E})$$

Computation.

$$\frac{\Gamma, x : \mathrm{Nat} \vdash A : \mathrm{Type}}{\Gamma \vdash u : A[\mathrm{Z}/x] \qquad \Gamma, x : \mathrm{Nat}, y : A \vdash v : A[\mathrm{S}(x)/x]}{\Gamma \vdash \mathrm{rec}(\mathrm{Z}, x \mapsto A, u, xy \mapsto v) = u : A[\mathrm{Z}/x]} \ (\mathrm{Nat_C^Z})$$

$$\frac{\Gamma \vdash t : \mathrm{Nat} \qquad \Gamma, x : \mathrm{Nat} \vdash A : \mathrm{Type} \qquad \Gamma \vdash u : A[\mathrm{Z}/x] \qquad \Gamma, x : \mathrm{Nat}, y : A \vdash v : A[\mathrm{S}(x)/x]}{\Gamma \vdash \mathrm{rec}(\mathrm{S}(t), x \mapsto A, u, xy \mapsto v) = v[t/x, \mathrm{rec}(t, x \mapsto A, u, xy \mapsto v)/y] : A[\mathrm{S}(t)/x]} \ (\mathrm{Nat_C^S})$$

Uniqueness.

$$\frac{\Gamma \vdash t : \mathrm{Nat}}{\Gamma \vdash \mathrm{rec}(t, x \mapsto \mathrm{Nat}, \mathrm{Z}, xy \mapsto \mathrm{S}(y)) = t : \mathrm{Nat}} \ (\mathrm{Nat_U})$$

In OCaml. The type Nat corresponds to the type

```
type nat =
  | Z
  | S of nat
```

where the constructors Z and S respectively correspond to Z and S, and the eliminator $\mathrm{rec}(t, x \mapsto A, u, xy \mapsto v)$ to

```
let rec ind t =
  match t with
  | Z -> u
  | S x -> let y = ind x in v
```

The computation rule says that

```
ind Z
```

reduces to u and

```
ind (S t)
```

reduces v where x has been replaced by t and y by ind t, and the uniqueness rule says that

```
let rec ind t =
  match t with
  | Z -> Z
  | S x -> let y = ind x in S y
```

is the identity function.

8.3.8 Other type constructors. There are two fundamental type constructions which were not given in this section: inductive types are presented in section 8.4 and identity types are presented in section 9.1.

8.4 Inductive types

We now present how to formalize general inductive types in type theory. We have already seen lots of examples in Agda in sections 6.4 and 6.5. For instance, the type of booleans is

```
data Bool : Set where
  false : Bool
  true  : Bool
```

the type of natural numbers is

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

the type of (rooted planar) binary trees is

```
data BTree : Set where
  leaf : BTree
  node : BTree → BTree → BTree
```

the type of (rooted planar) trees is

```
data Tree : Set where
  nil  : Tree
  node : List Tree → Tree
```

the type of lists is

```
data List (A : Set) : Set where
  nil  : List A
  cons : A → List A → List A
```

the type of vectors is

```
data Vec (A : Set) : ℕ → Set where
  nil  : Vec A zero
  cons : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

the type of finite sets is

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

ans so on.

8.4.1 W-types. The study and formalization of inductive types is notoriously difficult and a source of bugs and inconsistencies. For simplicity, we begin by studying a very restricted form of inductive types A, called *polynomial types* or *W-types*, which are defined in such a way that each constructor takes a finite number of arguments of type A (we will see that we can also easily generalize this to accept arguments whose type do not involve A, the most important part of the restriction is that constructors cannot have arguments whose type involve A in non-trivial ways, such as having an argument of type $A \to A$). In pseudo-Agda code, such a type would be defined as

```
data A : Set where
  C₁ : A → ... → A → A
  C₂ : A → ... → A → A
  ...
  Cₙ : A → ... → A → A
```

where `A` is the inductive type and the `Cᵢ` are the constructors. For instance, the type `Bool` of booleans, `Nat` of natural numbers and the type `BTree` of binary trees are of this form. In particular, the type `BTree` has two constructors (`leaf` and `node`), respectively taking 0 and 2 arguments.

Such a type is entirely characterized by

- a number n of constructors, and

- a function $f : \{0, \ldots, n-1\} \to \mathbb{N}$ which to i associates the number of arguments of the i-th constructor.

For instance,

- for booleans, we have $n = 2$ and $f(0) = f(1) = 0$,

- for natural numbers, we have $n = 2$, $f(0) = 0$ and $f(1) = 1$ (the 0-th and 1-st constructors are respectively zero and successor),

- for binary trees, we have $n = 2$, $f(0) = 0$ and $f(1) = 2$ (the 0-th and 1-st constructors are respectively leaf and node).

The problem with this data, namely the pair (n, f), is that it does not consist of types, and thus does not allow for very natural formalization in terms of typing rules. We will see below that it can however be encoded quite naturally into types.

Finite families of types. Suppose that our type theory contains the type $\bot$ with 0 element (section 8.3.1), the type $\top$ with 1 element (section 8.3.2) and coproducts (section 8.3.5). Given a natural number n, we can build a type Fin_n with n elements as

$$\mathrm{Fin}_n = \top + \top + \ldots + \top$$

the sum being $\bot$ in the case $n = 0$. For instance, the type Fin_4 with 4 elements is

$$\mathrm{Fin}_4 = \top + (\top + (\top + \top))$$

A typical element of this type is $\iota_r(\iota_r(\iota_l(\star)))$, but we will simplify the notations and write 0, 1, 2 and 3 for its elements. In Agda, we have already encountered

this type in section 6.4.8. It can be noted that, given a type A, defining a function $f : \mathrm{Fin}_n \to A$ precisely amounts to specifying n elements of A, those elements being $f(0), \ldots, f(n-1)$.

W-types. Now that we have made the previous remark, we can reformulate our definition of inductive types using types with a finite number of elements instead of natural numbers. A polynomial type consists of

– a type A with n elements, for some natural number n,

– for every element x of type A, a type $B(x)$ with n_x elements for some natural number n_x.

In other words, it consists of a pair (A, B), with

$$A : \mathrm{Type} \qquad\qquad B : A \to \mathrm{Type}$$

such that $A = \mathrm{Fin}_n$ for some natural number n and, for every $x : A$, we have $B(x) = \mathrm{Fin}_{n_x}$ for some natural number n_x. It turns out that this restriction to the case where A and $B(x)$ are finite types is not very useful in the following, so that we will drop it. Having an infinite type A (e.g. natural numbers) corresponds to having an infinite number of constructors, which seems worrying at first, but we will see that it is actually reasonable and useful.

Given a type A, and a type B which might have x as free variable, we write

$$\mathrm{W}(x : A).B$$

for the inductive type defined by this data and call it a *W-type*. Again, this should be thought of as an inductive type with a constructor for each element x of type A, this constructor taking as many arguments as there are elements in $B(x)$. The constructor W is binding x in B, and α-conversion allows us to rename it as we want.

Example 8.4.1.1. The type of binary trees can be defined by

$$A = \mathrm{Fin}_2 \qquad\qquad B(0) = \mathrm{Fin}_0 \qquad\qquad B(1) = \mathrm{Fin}_2$$

We now wonder what the terms of type $\mathrm{W}(x : A).B$ look like. Consider the type of binary trees as defined in Agda above. A typical element of this type is

```
node (node leaf (node leaf leaf)) leaf
```

which consists of a the constructor `node`, applied to two binary trees: the trees `node leaf (node leaf leaf)` and `leaf`. More generally, an element of the type $\mathrm{W}(x : A).B$ consists of

– a constructor, i.e. an element a of A, and

– n elements of $\mathrm{W}(x : A).B$, where n is the number of elements of the type $B\,a$, which are most naturally specified by giving a function $B\,a \to \mathrm{W}(x : A).B$.

W-types in Agda. The previous reformulation directly allows us to define *W*-types in Agda as follows:

```
data W (A : Set) (B : A → Set) : Set where
  sup : (a : A) → (B a → W A B) → W A B
```

The only constructor `sup` allows constructing an element of $W(x : A).B$ by specifying a constructor in A and arguments in the *W*-type, as explained above.

For instance, the type of natural numbers has two constructors, so that we can take A = `Bool` where, by convention, `false` corresponds to the constructor `zero` and `true` to `suc`. The first constructor takes zero arguments, which means that A `false` should be an empty type (we can take $\bot$) and A `true` takes one argument so that we should take `Arg true` to be a type with one element (we can take $\top$). We can thus define:

```
Nat : Set
Nat = W Bool (λ { false → ⊥ ; true → ⊤ })
```

Up to some syntactical heaviness (such as having to write booleans to call the constructors), this is precisely the usual inductive type for natural numbers. For instance, addition can be programmed "as usual":

```
_+_ : Nat → Nat → Nat
sup false _ + n = n
sup true x + n = sup true (λ { tt → x tt + n })
```

Similarly, the type of binary trees is

```
BTree : Set
BTree = W Bool (λ { false → ⊥ ; true → Bool })
```

Encoding into W-types. The class of types which we can handle looks quite restricted because the arguments of constructors can only be of the W-type itself. It is actually not, thanks to the extra generality brought by the possibility of having arbitrary type as A and $B(x)$, and not only finite types. For instance, the type of lists

```
data List (A : Set) : Set where
  nil  : List A
  cons : A → List A → List A
```

is not obviously a W-type because the constructor `cons` takes an argument of type A, whereas we are trying to define `List A`, and thus the arguments of constructors should have this type. However, instead of thinking of `cons` as one constructor, we can think of it as an infinite family of constructors `cons a`, one for each element `a` of A, each of which is taking one argument of type `List A`. In this way, it is natural to take `Maybe A` as the type of constructors where `nothing` corresponds to the constructor `nil` and `just a` corresponds to `cons a`, and we define

```
List : (A : Set) → Set
List A = W (Maybe A) (λ { nothing → ⊥ ; (just x) → ⊤ })
```

8.4.2 Rules for W-types. In order to add support for W-types, one should add the following constructions to expressions:

$$e ::= \ldots \mid \mathrm{W}(x : e).e' \mid \sup(e, e') \mid \mathrm{Wrec}(e, x \mapsto e', xyz \mapsto e'')$$

where $\mathrm{W}(x : A).B$ the W-type constructor, $\sup(t, u)$ constructs an element of a W-type with t as constructor and u as function specifying arguments, and $\mathrm{Wrec}(t, x \mapsto C, xyz \mapsto u)$ eliminates an element t of a W-type and produces an element of type C.

Formation.

$$\frac{\Gamma, x : A \vdash B : \mathrm{Type}}{\Gamma \vdash \mathrm{W}(x : A).B : \mathrm{Type}} \ (W_{\mathrm{F}})$$

Introduction.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/x] \to \mathrm{W}(x : A).B}{\Gamma \vdash \sup(t, u) : \mathrm{W}(x : A).B} \ (W_{\mathrm{I}})$$

Elimination.

$$\frac{\Gamma \vdash t : \mathrm{W}(x : A).B \qquad \Gamma, x : \mathrm{W}(x : A).B \vdash C : \mathrm{Type}}{\Gamma, x : A, y : B \to \mathrm{W}(x : A).B, z : \Pi(w : B).C[(y\,w)/x] \vdash u : C[\sup(x, y)/x]}{\Gamma \vdash \mathrm{Wrec}(t, x \mapsto C, xyz \mapsto u) : C[t/x]} \ (W_{\mathrm{E}})$$

Computation.

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : \mathrm{W}(x : A).B \vdash C : \mathrm{Type} \qquad \Gamma \vdash u : B[t/x] \to \mathrm{W}(x : A).B}{\Gamma, x : A, y : B \to \mathrm{W}(x : A).B, z : \Pi(w : B).C[(y\,w)/x] \vdash v : C[\sup(x, y)/x]}{\Gamma \vdash \mathrm{Wrec}(\sup(t, u), x \mapsto C, xyz \mapsto v) = v[t/x, u/y, \lambda w.\, \mathrm{Wrec}(u\,w, x \mapsto C, xyz \mapsto v)/z] : C[\sup(t, u)/x]} \ (W_{\mathrm{C}})$$

Uniqueness. This is not usually considered and requires function extensionality.

8.4.3 More inductive types. W-types are very fine if you want to perform a clean and easy implementation of inductive types, or want to study metatheoretic properties of types. In practice, proof assistants have more involved implementations of inductive types. One reason is user-friendliness: we want to be able to give nice names for constructors, have a nice syntax for pattern matching, generate pattern-matching cases automatically, etc. Also, we do not want the user to have to explicitly encode his types into W-types, and more generally we want to implement extensions of W-types. The interested reader is advised to look at good descriptions of actual inductive types in Agda [Nor07], in Coq [PM93] or theory [Dyb94]. We list below some common extensions of inductive types.

Indexed W-types. A first generalization of the notion of W-type is the support for indices. For instance, the type of finite sets is defined as

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

so that `Fin` n is a type with n elements. Here, the type takes a natural number n as argument, and various values for this argument are needed for constructors, e.g. `suc` needs an argument of type `Fin` n to produce a `Fin` $(n+1)$.

The definition of W-types can be modified in order to account for indices as follows. We only give here the implementation in Agda:

```
data W (I : Set) (A : I → Set) (B : (i : I) → A i → I → Set) : I → Set
  where
  sup : (i : I) (a : A i) → ((j : I) → B i a j → W I A B j) → W I A B i
```

In this type, `I` is the type for indices, `A i` is the type indicating the constructors with index `i`, and `B a j` indicates the number of arguments of index `j` of the constructor `a`.

Example 8.4.3.1. For instance, in the case of `Fin`,

  – `I` is the type of natural numbers,

  – `A 0` is the empty type ⊥ (there is no constructor for `Fin 0`) and, for `i > 0`, `A i` is the type `Bool` with two elements (there are two constructors for `Fin i`: respectively `zero` and `suc`),

  – for indices `i` and `j`,

    – the constructor `zero` of type `Fin j` takes zero argument of type `Fin i`

    – the constructor `suc` of type `Fin j` takes one argument of type `Fin i` when `suc i` is `j`, and zero argument otherwise,

  which determines the types `B i a j`.

We thus define the type `A` as

```
A : ℕ → Set
A zero    = ⊥
A (suc n) = Bool
```

the type `B` as

```
B : (n : ℕ) → A n → ℕ → Set
B (suc n) false m = ⊥
B (suc n) true m with n ≟ m
B (suc n) true m | yes _ = ⊤
B (suc n) true m | no  _ = ⊥
```

and finally, the type of finite sets as

```
Fin : ℕ → Set
Fin n = W ℕ A B n
```

Exercise 8.4.3.2. Define the types `Vec A n` of vectors of length `n` containing elements of type `A` using indexed W-types.

Mutually inductive types. One might want to define two inductive types which mutually depend on each other. For instance, trees and forest can be defined in a mutually inductive fashion as follows:

```
data Tree   : Set
data Forest : Set

data Tree where
  leaf : Tree
  node : Forest → Tree

data Forest where
  nil  : Forest
  cons : Tree → Forest → Forest
```

A tree takes a forest as argument and a forest is a list of trees (although we do not use the inductive type for lists here and define a new one adapted to forests).

Nested inductive types. One might want to define inductive types in which arguments are other inductive types applied to the type itself. For instance, trees can also be defined as nodes taking lists of trees as argument, lists being themselves defined as an inductive types:

```
open import Data.List

data Tree : Set where
  nil  : Tree
  node : List Tree → Tree
```

Inductive-inductive types. One might want to define both

  – an inductive type A and

  – a predicate on A (i.e. a function $A \to \text{Type}$)

whose definitions mutually depend on each other. For instance, the type of sorted lists can be defined along the predicate `_≤*_` (where x `≤*` l means that x is below every element of the list l, see section 6.7.2) as follows. In Agda, we first have to declare the type of the two definitions by

```
data SortedList : Set
data _≤*_        : ℕ → SortedList → Set
```

and we can then define both types by mutual induction by

```
data SortedList where
  empty : SortedList
  cons  : (x : ℕ) (l : SortedList) (le : x ≤* l) → SortedList

data _≤*_ where
  ≤*-empty : {x : ℕ} → x ≤* empty
  ≤*-cons  : {x y : ℕ} {l : SortedList} →
             x ≤ y → (le : y ≤* l) → x ≤* (cons y l le)
```

see figure 6.3 for an application of those definitions. Such types are called *inductive-inductive types* [FS12].

Coinductive types. Inductive types are defined as a smallest fixpoint, see section 1.3.3. For instance, the type of natural numbers is the smallest type containing zero and closed under successor. It is also possible to consider greatest fixpoints, and the resulting types are called *coinductive types*.

8.4.4 The positivity condition. When adding more general forms of inductive types, one should be very careful. Adding seemingly useful or natural inductive types can make the system inconsistent.

Inconsistent inductive types. As an illustration, consider inductive types where the arguments of constructors are types built from basic types, the inductive type we are defining, and arrows. For instance, with this formalism, the type of binary trees could be implemented in Agda as

```
data BTree : Set where
  leaf : BTree
  node : (Bool → BTree) → BTree
```

where the argument of the node constructor is `Bool → BTree` which is an arrow from a basic type (`Bool`) to the inductively defined type (`BTree`): given a function f of this type, f `false` indicates the first child and f `true` indicates the second child of the node.

Such inductive types also allow for a very natural implementation of λ-terms. Namely, since Agda already implements λ-calculus (α-conversion, β-reduction, etc.), we would like to use this instead of explicitly redefining those. One way to do this is to observe that the only thing we can do with an abstraction $\lambda x.t$ is to β-reduce it, and therefore implement it as the function which to a λ-term u associates the term $t[u/x]$, this is *normalization by evaluation* which is detailed in section 3.5.2. This suggests implementing λ-terms as the type

```
data Term : Set where
  abs : (Term → Term) → Term
```

(we should also add a constructor for variables, which we did not do here since it will play no role in the following explanation) and application as

```
app : Term → Term → Term
app (abs f) t = f t
```

However, remembering the course about λ-calculus in section 3.2.6, we start feeling bad because we remember that we can define a looping λ-term as

```
loop : Term
loop = app ω ω
```

where ω is defined as

```
ω : Term
ω = abs (λ x → app x x)
```

which contradicts the postulate that all terms should be terminating in Agda. Indeed, if we consider the small variation where we define terms

```
data Term : Set where
  abs : (Term → ⊥) → Term
```

then `app` has type `Term → Term → ⊥` and `loop` is a proof of ⊥, i.e. our logic is inconsistent!

The proof can further be simplified by defining

```
data Bad : Set where
  bad : (Bad → ⊥) → Bad
```

(we are simply giving another name to the new `Term` here), which is now thought of as a type equivalent to its own negation, thus allowing to prove ⊥. Namely, we can show the negation of this type by

```
not-Bad : Bad → ⊥
not-Bad (bad f) = f (bad f)
```

we construct a proof of the type

```
is-Bad : Bad
is-Bad = bad not-Bad
```

and thus conclude to an inconsistency:

```
absurd : ⊥
absurd = not-Bad is-Bad
```

The positivity condition. In practice, when defining the type `Bad` in Agda, we get an error message stating that

```
Bad is not strictly positive, because it occurs to the left of an
arrow in the type of the constructor bad in the definition of Bad.
```

This message indicates that our type is rejected, thus preventing the logic from being inconsistent, because it does not satisfy the "strict positivity condition" explained below. In order to test the above examples, you can however disable this check by writing

```
{-# NO_POSITIVITY_CHECK #-}
```

just before the definition of the type.

In order to build intuition, first consider traditional functions between sets. We write $A \Rightarrow B$ for the set of all functions from a set A to a set B. Given sets A, B and B', it can be noted that

$$B \subseteq B' \qquad \text{implies} \qquad (A \Rightarrow B) \subseteq (A \Rightarrow B')$$

Namely, given a function $f : A \to B$, the image of every element of A is an element of B and thus of B', i.e. f is a function from A to B'. However, on the left of arrows, the situation is reversed: for sets A, A' and B, we rather have

$$A \subseteq A' \qquad \text{implies} \qquad (A \Rightarrow B) \supseteq (A' \Rightarrow B)$$

Namely, any function defined for every element of A' is in particular defined for every element of A. Because of this behavior, the arrow types are said to be *covariant* in B and *contravariant* in A; we also say that A varies *negatively* and B varies *positively* in $A \Rightarrow B$. Traditionally, inductive types are obtained by "adding elements" to the type. For instance, natural numbers contain zero and for every natural number n, we add a new natural number, its successor. Now, if some constructors have negative occurrences of the inductively defined type, when adding more elements we should also remove some elements, because the constructor is contravariant, and the meaning of the inductively defined type is not clear at all. In terms of the formalization described in section 1.3.3, this means that the function induced by the description of the inductive type might not be increasing, so that we have no guarantee that it should have a smallest fixpoint.

The polarity (positive or negative) of a type can be defined as follows. For simplicity, we consider types of the form

$$A, B ::= X \mid A \to B$$

consisting either of a variable or an arrow. Given a type A, the polarity of a type which is a subterm of A, is defined by induction on A by

- the polarity of A is positive,

- in a type $B \to C$, the polarity of C is the same as the polarity of $B \to C$,

- in a type $B \to C$, the polarity of B is the opposite of the polarity of $B \to C$.

In other terms, the polarity at toplevel is positive, stays the same when we go to the right of an arrow and changes when we go to the left of an arrow. An Agda formalization of this is given in figure 8.1. A type is *strictly positive* when it is positive and we did not encounter negative types when computing the polarity.

Example 8.4.4.1. For instance, in the type

$$A \to ((B \to C) \to (D \to E))$$

the types A, C and D are negative and B and E is positive. The syntactic tree of the type can be written as follows
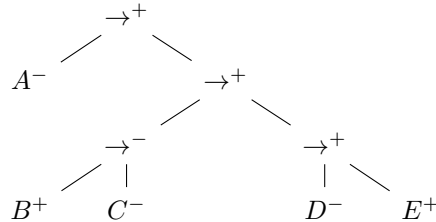
where $+$ or $-$ indicate the polarity of subtrees (positive or negative). The type E is strictly positive, but the type B is not strictly positive, because we computed its polarity by

- $A \to ((B \to C) \to (D \to E))$ is positive, thus

```
-- Polarities
data Polarity : Set where
  pos : Polarity
  neg : Polarity

-- Opposite of a polarity
op : Polarity → Polarity
op pos = neg
op neg = pos

-- Types
data Type : Set where
  var : ℕ → Type
  arr : Type → Type → Type

-- Subterm relation on types
data _<_ : Type → Type → Set where
  top   : {A : Type} → A < A
  left  : {A A' B : Type} → A < A' → A < arr A' B
  right : {A B B' : Type} → B < B' → B < arr A B'

-- Polarity of a type A in a type B
polarity : {A B : Type} → A < B → Polarity
polarity top       = pos
polarity (left p)  = op (polarity p)
polarity (right p) = polarity p
```

Figure 8.1: Polarities of types in Agda.

    – $(B \to C) \to (D \to E)$ is positive, thus

    – $B \to C$ is negative, thus

    – B is positive,

and we encountered negative types.

    Agda (and most other proof assistants such as Coq) implement the following restriction on inductive types: given a constructor of an inductive type A, if A occurs in the argument of an inductive type, then it must do so strictly positively. For instance, `Bad` above is rejected because the constructor `bad` takes one argument of type `Bad → ⊥`, where `Bad` occurs negatively. The above counter-example explains why we must forbid negative occurrences. The reason why we must further restrict to strictly positive occurrences is explained for Coq in [CP88]; its usefulness in Agda is not so clear [Coq13], which is why we did not provide a counter-example.

8.4.5 Disjointedness and injectivity of constructors. We present here two important properties of inductive type constructors in Agda related to equality.

Disjointedness of constructors. Constructors are *disjoint*, meaning that values made using two different constructors are necessarily different. For instance, over natural numbers, zero cannot be equal to the successor of some number:

```
zero-suc : {n : ℕ} → zero ≡ suc n → ⊥
zero-suc ()
```

Here, the empty pattern `()` means that Agda should check by himself that the case where `zero` is equal to `suc n` cannot happen, which it does thanks to the disjointedness assumption.

Injectivity of constructors. Constructors are *injective*, meaning that if two constructed values are equal then the arguments are equal. For instance:

```
suc-injective : {m n : ℕ} → suc m ≡ suc n → m ≡ n
suc-injective refl = refl
```

(this could also be shown directly using `cong`).

Injectivity of type constructors. Type constructors are not injective by default. For instance, the following does not typecheck:

```
list-inj : {A B : Set} → List A ≡ List B → A ≡ B
list-inj refl = refl
```

We can however explicitly ask Agda to make type constructors injective, by adding the following pragma at the beginning of the file:

```
{-# OPTIONS --injective-type-constructors #-}
```

The reason why it is not enabled by default is that it makes the system inconsistent together with the excluded middle, thus preventing from safely working in classical logic. A counter-example was found by Hur based on the following observation [Hur10]. We can define an inductive data type of the form

```
data I : (Set → Set) → Set where
```

(the constructors will not matter, so we might as well choose to have none). The injectivity of the type constructor `I` amounts to having an injection of `Set → Set` into `Set`, which is excluded by a diagonal argument à la Cantor, appendix A.4. Namely, with injective type constructors, we can show

```
inj : {x x' : Set → Set} → I x ≡ I x' → x ≡ x'
inj refl = refl
```

In order to use the Cantor diagonal argument formalized in appendix A.4.2, we have to show that `Set` contains two distinct types, say $\top$ and $\bot$,

```
⊤≢⊥ : ¬ (⊤ ≡ ⊥)
⊤≢⊥ p = subst (λ A → A) p tt
```

and suppose that the law of excluded middle holds

```
postulate lem : (A : Set₁) → Dec A
```

We finally conclude

```
absurd : ⊥
absurd = Cantor.no-injection ⊤≢⊥ lem I inj
```

8.5 Implementing type theory

We now explain how to implement a typechecker in dependent type theory in a reasonably efficient and principled way. We chose to implement a type theory with Π-types, natural numbers and identity types in order to show most of the principles needed in order to implement a typechecker. Identity types will be presented in section 9.1, and you can simply ignore them if you have not yet read this part of the book. We could have more generally implemented inductive types (or, at least, W-types) in a similar way [CKNT09], but felt that the code would be more readable when specialized to natural numbers and identity types. The version given here is a variant of the standard implementations for dependent types [Coq96, GL02, CKNT09, LMS10, Bau12].

The basic idea is to implement a bidirectional typechecking algorithm, similar to the one we already presented for simply-typed λ-calculus in section 4.4.5: we try to check that a term has a given type when we have a candidate for the type, otherwise we try to infer the type of the term. The reason for this is that we generally declare the type of functions before defining them, but do not want to annotate each λ-abstraction with the type of the variable. This is for instance the way things are in Agda. There is a subtlety though: when comparing expressions (terms or types), we should do so modulo $\alpha\beta$-convertibility. As detailed in section 4.2.4, by the confluence and termination of the calculus, we can decide whether two expressions t and u are convertible, by computing their normal forms (i.e. β-reducing them as much as we can) and checking the resulting terms for α-convertibility. This means that we should choose a way to implement β-reduction among the ones presented in section 3.5. The normalization by evaluation technique (section 3.5.2) is the most suitable for us: it is quite easy to implement because it relies on the implementation of the β-reduction of

the programming language (OCaml in our case). Moreover, it allows for an efficient implementation of convertibility: instead of fully performing β-reduction, we can compute weak head normal forms, so that we can potentially detect when two terms are not equal without fully reducing them.

8.5.1 Expressions. We begin by formally defining expressions as

```
type expr =
  | Var of string                     (** a variable *)
  | Abs of string * expr        (** a lambda-abstraction *)
  | App of expr * expr             (** an application *)
  | Pi of string * expr * expr        (** a Pi-type *)
  | Type of int                      (** a universe *)
  | Nat                   (** type of natural numbers *)
  | Zero                               (** zero *)
  | Succ of expr                   (** successor *)
  | Ind of expr * expr * expr * expr     (** induction *)
  | Id of expr * expr * expr      (** identity type *)
  | Refl of expr                  (** reflexivity *)
  | J of expr * expr * expr * expr * expr * expr (** id elim *)
```

An expression is thus either a variable, a λ-abstraction

$$\lambda x.t \qquad \text{written} \qquad \texttt{Abs(x, t)}$$

an application of an expression to another, a Π-type

$$\Pi(x : a).b \qquad \text{written} \qquad \texttt{Pi(x, a, b)}$$

a universe of given level, the type of natural numbers, zero, the successor of a natural number, the induction principle

$$\text{rec}(n, x \mapsto A, z, mr \mapsto s)$$

written

$$\texttt{Ind(n, Abs(x, a), z, Abs(m, Abs(r, s)))}$$

(note that we use abstractions as arguments of `Ind` in order to avoid having to handle α-conversion here, and only take care of it for abstractions, see section 4.3.3) and identity type

$$\text{Id}_A(t, u) \qquad \text{written} \qquad \texttt{Id(a, t, u)}$$

a reflexivity proof

$$\text{refl}(t) \qquad \text{written} \qquad \texttt{Refl(t)}$$

or a J eliminator

$$\text{J}(e, xye \mapsto A, x \mapsto r)$$

written

$$\texttt{J(a, Abs(x, Abs(y, Abs(e, a))), Abs(x, r), t, u, e)}$$

Values. A term will evaluate to a *value* which is, by definition, a term which does not reduce anymore. The type corresponding to values is

```
type value =
  | VAbs of (value -> value)      (** a lambda-abstraction *)
  | VPi of value * (value -> value)        (** a Pi-type *)
  | VType of int                        (** a universe *)
  | VNat                  (** type of natural numbers *)
  | VZero                              (** zero *)
  | VSucc of value                   (** successor *)
  | VId of value * value * value      (** identity type *)
  | VRefl of value                 (** reflexivity *)
  | VNeutral of neutral          (** a neutral value *)
```

which roughly corresponds to the definition of expressions, with a few notable differences, as we now explain. For abstractions (`VAbs`), the body is not yet evaluated, because we are computing weak head normal forms: instead, we have a function which given an argument, will compute the normal form of the body with the argument substituted as expected. Similarly, a Π-type $\Pi(x : A).B$ is stored in `VPi` as the type A and the function $\lambda x^A.B$, which provides the type B given the argument of type A. The last case corresponds to *neutral values*: those are expressions in which the computation is not fully performed, but is stuck because we do not know the value for some variable. For instance, given a variable x and a term t, the term $x\,t$ is a value: in order to evaluate this application, we would need to know the value for x, which should be a λ-abstraction. Neutral values are defined by the type

```
and neutral =
  | NVar of string
  | NApp of neutral * value
  | NInd of neutral * value * value * value
  | NJ   of value * value * value * value * value * neutral
```

and thus consist either of a variable, or a neutral value applied to a value (e.g. $x\,t$), or an induction on a neutral value (e.g. an induction on a variable) or an elimination of a neutral proof of identity.

8.5.2 Evaluation. We can then easily write a function which applies a value t to another value u. In the case t is an abstraction, we apply it to u. Otherwise, if we assume that the terms are suitably typed, t has to be a neutral value (e.g. we cannot apply a natural number to some other term), in which case the result is still a neutral value:

```
let vapp u v =
  match u with
  | VAbs f     -> f v
  | VNeutral t -> VNeutral (NApp (t, v))
  | _          -> assert false
```

Thanks to this helper function, we can write a function `eval` which evaluates an expression `t` to a value. The function also takes an environment `env`, which is a list of pairs associating to a free variable its value, in the case it is known.

```
let rec eval env t =
  match t with
  | Var x        ->
    (try List.assoc x env with Not_found -> VNeutral (NVar x))
  | Abs (x, e)   -> VAbs (fun v -> eval ((x,v)::env) e)
  | App (e1, e2) -> vapp (eval env e1) (eval env e2)
  | Pi (x, a, e) -> VPi (eval env a, fun v -> eval ((x,v)::env) e)
  | Type i       -> VType i
  | Nat          -> VNat
  | Zero         -> VZero
  | Succ e       -> VSucc (eval env e)
  | Ind (n, a, z, s) ->
    let n = eval env n in
    let a = eval env a in
    let z = eval env z in
    let s = eval env s in
    let rec f = function
      | VZero      -> z
      | VSucc n    -> vapp (vapp s n) (f n)
      | VNeutral n -> VNeutral (NInd (n, a, z, s))
      | _          -> assert false
    in
    f n
  | Id (a, t, u) -> VId (eval env a, eval env t, eval env u)
  | Refl t -> VRefl (eval env t)
  | J (a, p, r, t, u, e) ->
    (
      match eval env e with
      | VRefl _    -> eval env r
      | VNeutral e ->
        VNeutral (NJ (eval env a, eval env p, eval env r,
                      eval env t, eval env u, e))
      | _          -> assert false
    )
```

As explained above, when evaluating a function (`Abs`), we return a function which will return the value corresponding to the body, provided the argument, which is stored in the environment. We use the function `vapp` in order to evaluate applications (`App`). For constructors corresponding to types and introduction rules, the function simply consists in evaluating all the arguments of the constructor. For the constructors corresponding to elimination rules, we evaluate the argument we are eliminating and then evaluate the construction accordingly. For instance, for induction (`Ind`), we evaluate the natural number n on which the induction is applied and compute the result of the induction accordingly, depending on whether the result is zero, a successor, or a variable.

8.5.3 Convertibility. Our goal is now to decide the convertibility of expressions. As explained above, this is basically performed by evaluating expressions to values and then comparing the resulting values for equality (we call `veq` the function which compares two values). However, since values may contain

functions (under the `VAbs` constructors), we first have to implement a *readback* function which will convert a value into an expression, following the same techniques as in section 3.5.2.

Readback. The readback function takes as arguments an natural number `k` (to generate fresh variables) and a value, and produces an expression:

```
let rec readback k v =
  let rec neutral k = function
    | NVar x ->
      Var x
    | NApp (t, u) ->
      App (neutral k t, readback k u)
    | NInd (n, a, z, s) ->
      Ind (neutral k n, readback k a, readback k z, readback k s)
    | NJ (a, p, r, t, u, e) ->
      J (readback k a, readback k p, readback k r,
         readback k t, readback k u, neutral k e)
  in
  match v with
  | VAbs f ->
    let x = fresh k in
    Abs (x, readback (k+1) (f (var x)))
  | VPi (a, b) ->
    let x = fresh k in
    Pi (x, readback k a, readback (k+1) (b (var x)))
  | VType i -> Type i
  | VNat -> Nat
  | VZero -> Zero
  | VSucc n -> Succ (readback k n)
  | VId (a, t, u) ->
    Id (readback k a, readback k t, readback k u)
  | VRefl t -> Refl (readback k t)
  | VNeutral t -> neutral k t
```

This function essentially consists in translating the constructors of `value` into the corresponding constructors of `expr`. The only subtlety can be found in the case of `VAbs` (and `VPi` which is similar). In order to generate the expression corresponding to an abstraction `VAbs f`, we apply `f` to a fresh variable, whose name is generated thanks to the natural number `k`. Namely, we use the following helper function to construct a "fresh" variable name with index k:

```
let fresh k = "x@"^string_of_int k
```

(we suppose that the user will never input a variable name containing the character `@`). Above, the function `var` is a shorthand to construct a variable with given name:

```
let var x = VNeutral (NVar x)
```

Equality of values. Because of the way the readback function is implemented, by canonically generating variable names when needed using an index `k`, two values

will be α-convertible when they have the same readback. We can therefore test the equality of two values t and u with the following function:

```
let veq k t u = readback k t = readback k u
```

More efficient equality. The above test for equality of values is not very efficient: it essentially requires evaluating the whole term, which can be very costly, whereas this is unnecessary when the two terms are not equal. For instance, the two terms VAbs f and VZero are not equal, and there is no need to proceed to the evaluation of f in order to determine this. The following refined test for equality takes this into account: it combines both readback and comparison, and amounts to computing the weak head normal forms of the two terms (see section 3.5.1) in order to compare them, and only evaluating under abstractions if the two weak head normal forms are abstractions.

```
let rec veq k t u =
  let rec neq k t u =
    match t, u with
    | NVar x, NVar y -> x = y
    | NApp (t, v), NApp (t', v') ->
      neq k t t' && veq k v v'
    | NInd (n, a, z, s), NInd (n', a', z', s') ->
      neq k n n' && veq k a a' && veq k z z' && veq k s s'
    | NJ (a, p, r, t, u, e), NJ (a', p', r', t', u', e') ->
      veq k a a' && veq k p p' && veq k r r' &&
      veq k t t' && veq k u u' && neq k e e'
    | _, _ -> false
  in
  match t, u with
  | VAbs f, VAbs g ->
    let x = var (fresh k) in
    veq (k+1) (f x) (g x)
  | VPi (a, b), VPi (a', b') ->
    let x = var (fresh k) in
    veq k a a' && veq (k+1) (b x) (b' x)
  | VType i, VType j -> i = j
  | VNeutral t, VNeutral u -> neq k t u
  | VNat, VNat -> true
  | VZero, VZero -> true
  | VSucc t, VSucc u -> veq k t u
  | VId (a, t, u), VId (a', t', u') ->
    veq k a a' && veq k t t' && veq k u u'
  | VRefl t, VRefl u -> veq k t u
  | _, _ -> false
```

The helper function neq compares neutral values for equality.

Exercise 8.5.3.1. Modify this function in order to compare values for η-equivalence. You should start by adding a new argument to the function which is the common type of the two values. See also exercise 7.5.3.1.

8.5.4 Typechecking. Finally, we can implement a type inference function `infer` as follows. We follow here the principles of bidirectional typechecking and define it at the same time (by mutual recursion) as one performing type checking, i.e. this is quite similar to the developments of section 4.4.5. The type inference function takes as argument an index `k` for generating fresh variables as above, a typing environment `tenv` associating to variable names a type, an environment `env` associating to variable names a value, and a term `t` whose type we would like to determine. This function is essentially, a translation to OCaml of the natural deduction rules of sections 8.1, 8.3 and 9.1.3:

```
let rec infer k tenv env t =
  match t with
  | Var x ->
    (
      try List.assoc x tenv
      with Not_found -> raise (Unbound_variable x)
    )
  | App (t, u) ->
    (
      match infer k tenv env t with
      | VPi (a, b) ->
        check k tenv env u a;
        b (eval env u)
      | _ -> raise Type_error
    )
  | Pi (x, a, b) ->
    let i = universe k tenv env a in
    let a = eval env a in
    let j = universe k ((x,a)::tenv) env b in
    VType (max i j)
  | Type i -> VType (i+1)
  | Nat -> VType 0
  | Zero -> VNat
  | Succ t ->
    check k tenv env t VNat;
    VNat
  | Ind (n, a, z, s) ->
    (
      check k tenv env n VNat;
      match eval env a with
      | VPi (VNat, a) ->
        let n = eval env n in
        check k tenv env z (a VZero);
        check k tenv env s
          (VPi (VNat, fun n -> varr (a n) (a (VSucc n))));
        a n
      | _ -> raise Type_error
    )
  | Id (a, t, u) ->
    let i = universe k tenv env a in
```

```
    let a = eval env a in
    check k tenv env t a;
    check k tenv env u a;
    VType i
  | Refl t ->
    let a = infer k tenv env t in
    let t = eval env t in
    VId (a, t, t)
  | J (a, p, r, t, u, e) ->
    let i = universe k tenv env a in
    let a = eval env a in
    check k tenv env p
      (VPi (a, fun x ->
            VPi (a, fun y -> varr (VId (a, x, y)) (VType i))));
    let p = eval env p in
    let p x y e = vapp (vapp (vapp p x) y) e in
    check k tenv env r (VPi (a, fun x -> p x x (VRefl x)));
    check k tenv env t a;
    check k tenv env u a;
    let t = eval env t in
    let u = eval env u in
    check k tenv env e (VId (a, t, u));
    let e = eval env e in
    p t u e
  | Abs _ -> raise Type_error
```

This function raises an error `Unbound_variable` when an undeclared variable is used and `Type_error` when the expression does not typecheck. It uses the following helper function to construct an arrow type, as a non-dependent Π-type:

```
let varr a b = VPi (a, fun _ -> b)
```

This function is defined by mutual induction with a function which checks that an expression is a type and returns its universe level:

```
and universe k tenv env t =
  match infer k tenv env t with
  | VType i -> i
  | _       -> raise Type_error
```

and with a function which checks that a term `t` has a given type `a`:

```
and check k tenv env t a : unit =
  match t, a with
  | Abs (x, t), VPi (a, b) ->
    let y = var (fresh k) in
    check (k+1) ((x,a)::tenv) ((x,y)::env) t (b y)
  | Refl t, VId (_, u, v) ->
    let t = eval env t in
    if not (veq k t u) then raise Type_error;
    if not (veq k t v) then raise Type_error
```

```
  | t, a ->
    let a' = infer k tenv env t in
    if not (veq k a a') then raise Type_error
```

Note that the case where the term is an abstraction $\lambda x.t$ (constructor `Abs`) and the type is a Π-type $\Pi(y : A).B$ (constructor `VPi`) is subtle: when checking that the body t has type B, we do so by after replacing both x and y by a fresh variable name.

8.5.5 Testing. In order to test our implementation, we can check that the addition has the type $A = \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}$:

```
let () =
  let a = varr VNat (varr VNat VNat) in
  let t =
    Abs (
      "m",
      Ind (
        Var "m",
        Pi ("_", Nat, Pi ("_", Nat, Nat)),
        Abs ("n", Var "n"),
        Abs ("m",
             Abs ("r",
                  Abs ("n", Succ (App (Var "r", Var "n")))))
      )
    )
  in
  check 0 [] [] t a
```

Of course, it is not reasonable to proceed in this way in order to use the implementation and one should implement a proper lexer and parser. We do not describe this part here since it is out of the scope of this book.

Homotopy type theory

In the introduction of chapter 2, we have motivated the exploration of intuitionistic logic by changing the intuitive meaning we give to types: instead of thinking of them as booleans, it is much more satisfactory to consider that they should be interpreted as sets, where it makes sense to consider various elements of a type. Namely, the boolean interpretation is too limited because when a type A is not false (i.e. empty) there is only one reason why this could be: A is necessarily true (i.e. the set with one element) and in this case there is only one proof of A (the only element of the set). In this sense, the boolean interpretation does not allow for considering the possibility that a type should admit various proofs. Now, if we try to make sense of equality in type theory, we discover that the set-theoretic interpretation of logic suffers from the same limitations. Namely, in a set, when two elements x and y are equal there is only one reason why this could possibly be: this is because x is the same as y.

This suggests changing once again the semantics we give to types and interpret them, not as booleans, not as sets, but as *spaces*. In this interpretation, proofs of equality correspond to paths, and we can thus conceive of models where there can be various ones. Homotopy type theory is dependent type theory seen from this point of view, and was introduced by Awodey and Voevodsky in the 2000's. The latter discovered that an additional axiom, called *univalence*, was required for the logic to match the situation in spaces, and homotopy type theory is usually understood with this axiom.

This makes the mathematician happy because he discovers that logic is secretly all about geometry. We will not dive too far in this direction because this would require introducing too much material and this is already wonderfully covered in [Uni13]. This also should make the computer scientist happy, because it allows for a clean handling of isomorphic data structures. Namely, it often occurs that we have the choice between various isomorphic ways of representing data, for instance lists or arrays, and we would like to automatically transfer the properties of one to the other. We will see that univalence allows this: two isomorphic types will be equal and we will thus be able to transport functions from one to the other.

The reader interested in learning more about the topic is urged to read the foundational book about the topic [Uni13]. The course notes of Altenkirch [Alt19] and Escardó [Esc19] are also very helpful.

We begin by introducing identity types in section 9.1, explain how types can be interpreted as spaces in section 9.2, discuss the classification of types as n-types in section 9.3, introduce the univalence axiom in section 9.4 and present higher inductive types in section 9.5.

9.1 Identity types

9.1.1 Definitional and propositional equality. In type theory, we have two notions of equality.

– The *definitional equality* states that some terms cannot be distinguished: this is the "=" relation in the inference rules, which corresponds to identifying terms under β-equivalence (or generalizations of it to terms with constructors other than λ-abstractions).

– The *propositional equality* or *identity* is a particular type expressing the fact that we consider two terms as equal.

In Agda, there is no notation for definitional equality, because there is simply no way to distinguish between two definitionally equal terms. On the other hand, `t ≡ u` expresses propositional equality between two terms `t` and `u`: we can provide a proof of such a fact and reason about it, but, when using `u` in place of `t`, we should perform some explicit manipulations (e.g. with `subst`) in order to explain to Agda that we can replace one by the other.

For instance, consider the usual definition of addition, see section 6.4.2:

```
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
```

The terms `zero + n` and `n` are definitionally equal: the second line in the above definition explicitly states that this should be the case. For this reason, the two can be used interchangeably and, for instance, we can give a vector of length `zero + n` where a vector of length `n` is expected. In contrast, the terms `n + zero` and `n` are not definitionally equal (there is no line in the definition of the addition which explicitly states that this should be the case), but we can show that they are propositionally equal, i.e. `n + zero ≡ n`, which requires reasoning on addition (by induction). This is the reason why the proof of left unitality of addition is so simple

```
+-zero' : (n : ℕ) → zero + n ≡ n
+-zero' n = refl
```

whereas the right unitality is more involved

```
+-zero : (n : ℕ) → n + zero ≡ n
+-zero zero    = refl
+-zero (suc n) = cong suc (+-zero n)
```

In this chapter, we mostly focus on propositional equality, leaving the definitional one implicit as it should be, and sometimes simply say *equality* for propositional equality. The propositional equality is also referred to as *identity* and a type `t ≡ u` as an *identity type*.

9.1.2 Propositional equality in Agda. We have already seen in section 6.6 that the definition of propositional equality is expressed in Agda with the following inductive type:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

It has only one constructor, `refl`, which expresses the reflexivity of equality: a term is equal to itself. In particular, two definitionally equal terms are propositionally so.

9.1.3 The rules. The rules for *propositional equality*, or *identity types*, follow from the above definition of equality as an inductive type, but can also be given directly, as for other connectives. These were first formulated by Martin-Löf [ML75, MLS84].

We extend the syntax of expressions with

$$e ::= \dots \mid \mathrm{Id}_e(e', e'') \mid \mathrm{refl}(e) \mid \mathrm{J}(e, xyz \mapsto e', x' \mapsto e'')$$

The new constructions are the following:

- the type $\mathrm{Id}_A(t, u)$ is called an *identity type* and expresses the fact that two terms t and u of type A are equal,

- $\mathrm{refl}(t)$ is the reflexivity of t, and

- J is the eliminator for identities.

In the following, we will often simply write $t \equiv u$ instead of $\mathrm{Id}_A(t, u)$, in accordance with Agda's notation for equality types.

Formation. The formation rule states that we can consider the type of propositional equalities, or identities, between any two terms t and u of the same type:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathrm{Id}_A(t, u) : \mathrm{Type}} \ (\mathrm{Id_F})$$

Introduction. The constructor refl allows proving the reflexivity of equality on a given term t:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathrm{refl}(t) : \mathrm{Id}_A(t, t)} \ (\mathrm{Id_I})$$

Elimination. The eliminator states that in order to prove a property B depending on a proof p that two terms t and u of type A equal, it is enough to give a proof r of it in the case where p is reflexivity:

$$\frac{\Gamma \vdash p : \mathrm{Id}_A(t, u) \qquad \Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash B : \mathrm{Type} \qquad \Gamma, x : A \vdash r : B[x/x, x/y, \mathrm{refl}(x)/z]}{\Gamma \vdash \mathrm{J}(p, xyz \mapsto B, x \mapsto r) : B[t/x, u/y, p/z]} \ (\mathrm{Id_E})$$

Computation. The computation rule expresses the fact that, when we use a proof constructed by J in the case where the considered proof of identity is reflexivity, we recover the proof r we provided:

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash B : \mathrm{Type} \qquad \Gamma, x : A \vdash r : B[x/x, x/y, \mathrm{refl}(x)/z]}{\Gamma \vdash \mathrm{J}(\mathrm{refl}(t), xyz \mapsto B, x \mapsto r) = r[t/x] : B[t/x, t/y, \mathrm{refl}(t)/z]} \ (\mathrm{Id_C})$$

Uniqueness. The uniqueness rule states that any term t depending on an identity z, can be obtained from its restriction to the case where z is the reflexivity, by using the J rule:

$$\frac{\Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash B : \mathrm{Type} \qquad \Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash t : B}{\Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash \mathrm{J}(z, xyz \mapsto B, x \mapsto t[x/y, \mathrm{refl}(x)/z]) = t : B} \ (\mathrm{Id}_\mathrm{U})$$

This uniqueness rule, which was present in Martin-Löf's original system, is debatable. In particular, it implies that the following rule, sometimes called *equality reflection*, which states two propositionally equal terms are definitionally so, is admissible:

$$\frac{\Gamma \vdash p : \mathrm{Id}_A(t, u)}{\Gamma \vdash t = u : A}$$

Namely, given a type A in a context Γ, we deduce, using the uniqueness rule,

$$\frac{\begin{array}{c} \Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash A : \mathrm{Type} \\ \Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash x : A \end{array}}{\Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash \mathrm{J}(z, xyz \mapsto A, x \mapsto x) = x : A} \ (\mathrm{Id}_\mathrm{U})$$

Similarly, we can also deduce, in the same context

$$\mathrm{J}(z, xyz \mapsto A, x \mapsto x) = y$$

and thus $x = y$ by transitivity, i.e. the following rule is admissible:

$$\frac{}{\Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash x = y : A}$$

We finally obtain the equality reflection rule by substituting x for t, y for u and z for p. In a similar way, one can show that the rule

$$\frac{}{\Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash z = \mathrm{refl}(x) : \mathrm{Id}_A(x, y)}$$

is admissible, i.e. reflexivity is the only possible proof of equality. A type theory allowing those rules is called *extensional*, and has the inconvenient property that its typechecking is undecidable [Hof95]. We will thus not postulate this rule in the following, and thus consider *intensional* type theory, for which typechecking can be mechanized. We will moreover see that not postulating that reflexivity is the only possible proof of equality allows for much richer models.

In Agda. The eliminator J corresponds to matching a proof of equality with `refl`:

```
J : {A : Set} {x y : A} (p : x ≡ y)
    (B : (x y : A) → x ≡ y → Set)
    (r : (x : A) → B x x refl)
    → B x y p
J {A} {x} {.x} refl B r = r x
```

Note that the second line corresponds precisely to the computation rule for identity.

In Agda, for reasons explained above, the uniqueness rule does not hold, but the variant expressed with propositional equality instead of definitional equality does:

```
J-η : {A : Set} {x y : A} (p : x ≡ y)
      (P : (x y : A) (p : x ≡ y) → Set)
      (t : (x y : A) (p : x ≡ y) → P x y p) →
      J p P (λ x → t x x refl) ≡ t x y p
J-η refl P t = refl
```

9.1.4 Leibniz equality. The definition of equality given above is not the first one one might think of. Another definition which is perhaps easier to accept was proposed by Leibniz [Lei86]. In this context, two things are said to be

- *identitical* when they are propositionally equal,

- *indiscernible* when a property satisfied by one is necessarily satisfied by the other.

There are two possible implications between those notions. The implication

$$\text{identical} \Rightarrow \text{indiscernible}$$

is called the principle of *indiscernability of identicals*. This is easy to take for granted: if two things x and y are equal then we should be able to replace an occurrence of x by y in every property. In other words, equality should be a congruence. The other implication

$$\text{indiscernible} \Rightarrow \text{identical}$$

is called the principle of *identity of indiscernibles*: it states that two things satisfying the same properties are the same. This is somewhat of an "interactive" point of view on the world, considering that in order for two things to be distinct, there should be some sort of experiment which allows distinguishing between the two. Leibniz postulated that both principles hold, i.e. the two notions are equivalent. The reference often quoted for the second principle is the following [Lei86]:

> *il n'est pas vray, que deux substances se ressemblent entierement et soyent differentes* solo numero

(which goes on with assertions such as "*On peut même dire que toute substance porte en quelque façon le caractere de la sagesse infinie et de la toute puissance de Dieu, et l'imite autant qu'elle en est susceptible.*" which are less clear from a logical point of view). If we also accept this implication, then we can in fact take indiscernability as a definition for equality. This is sometimes called *Leibniz equality*:

Definition 9.1.4.1 (Leibniz equality). Two things are equal when every property satisfied by one is also satisfied by the other.

We write $x \doteq y$ when x and y are equal according to Leibniz definition, i.e. when for every predicate $P(z)$, with a free variable z, we have

$$P(x) \Rightarrow P(y) \tag{9.1}$$

In Agda, this can be formalized in the following way:

```
_≐_ : {A : Set} → (x y : A) → Set₁
_≐_ {A} x y = (P : A → Set) → (P x → P y)
```

This relation is clearly reflexive:

```
≐-refl : {A : Set} {x : A} → x ≐ x
≐-refl P p = p
```

It is however not obvious that it is symmetric. In fact, our first inclination would have be to have taken $P(x) \Leftrightarrow P(y)$ in the definition (9.1), i.e. an equivalence instead of an implication, so that symmetry would be obvious. However, this is not necessary, because the converse implication can always be deduced:

Lemma 9.1.4.2. If $x \doteq y$ then $y \doteq x$.

Proof. Suppose that $x \doteq y$, and fix a predicate P. Consider the predicate $Q(z) = (P(z) \Rightarrow P(x))$. By definition of $x \doteq y$, we have $Q(x)$ implies $Q(y)$, i.e. $P(x) \Rightarrow P(x)$ implies $P(y) \Rightarrow P(x)$. But, $P(x) \Rightarrow P(x)$ is obviously true, so that we have $P(y) \Rightarrow P(x)$. Since, this holds for every predicate P, we have $y \doteq x$. □

In Agda, this can be formalized as follows:

```
≐-sym : {A : Set} {x y : A} → x ≐ y → y ≐ x
≐-sym {x = x} e P = e (λ z → (P z → P x)) (λ p → p)
```

Transitivity can be shown in a similar fashion:

```
≐-trans : {A : Set} {x y z : A} → x ≐ y → y ≐ z → x ≐ z
≐-trans {x = x} e e' P = e' (λ z → (P x → P z)) (e P)
```

Now, the question is how this definition of equality $\doteq$ compares to the propositional equality $\equiv$ of the previous section: if the two did not agree then we would have to discuss which one is the right one, which looks like a metaphysical debate. Fortunately, both of them can be shown to coincide. The fact that propositional equality implies Leibniz equality follows immediately by induction, since when $x \equiv y$, we can restrict to the case where x and y are the same (and the proof of $x \equiv y$ is reflexivity):

```
≡-to-≐ : {A : Set} {x y : A} → x ≡ y → x ≐ y
≡-to-≐ refl = ≐-refl
```

and the converse implication can be obtained as the variant of the proof that $\doteq$ is symmetric:

```
≐-to-≡ : {A : Set} {x y : A} → x ≐ y → x ≡ y
≐-to-≡ {x = x} e = e (λ z → x ≡ z) refl
```

More details can be found in [ACD⁺18].

9.1.5 Extensionality of equality. Two things are said to *extensionally equal* when their constituents are equal. We expect that equality coincides with extensional equality, and it is in fact so for inductively defined types.

Extensional equality on pairs. Two pairs are extensionally equal when their members are equal. It is easy to show that two extensionally equal pairs are equal:

```
×-≡ : {A B : Set} {x x' : A} {y y' : B} →
      x ≡ x' → y ≡ y' → (x , y) ≡ (x' , y')
×-≡ refl refl = refl
```

and conversely, two equal pairs are extensionally so:

```
≡-× : {A B : Set} {x x' : A} {y y' : B} →
        (x , y) ≡ (x' , y') → (x ≡ x') × (y ≡ y')
≡-× refl = refl , refl
```

Extensional equality on lists. Similarly, two lists are extensionally equal when they have the same (i.e. equal) elements. In Agda, this relation can be defined inductively by

```
data _==_ {A : Set} : (l l' : List A) → Set where
  ==-[] : [] == []
  ==-::  : {x x' : A} {l l' : List A} →
          x ≡ x' → l == l' → (x :: l) == (x' :: l')
```

This relation is easily shown to be reflexive by induction

```
==-refl : {A : Set} (l : List A) → l == l
==-refl [] = ==-[]
==-refl (x :: l) = ==-:: refl (==-refl l)
```

from which one can show that equality implies extensional equality:

```
≡-== : {A : Set} {l l' : List A} → l ≡ l' → l == l'
≡-== {l = l} refl = ==-refl l
```

Conversely, one can show that two lists with the same head and the same tail are equal:

```
≡-:: : {A : Set} → {x x' : A} → {l l' : List A} →
       x ≡ x' → l ≡ l' → x :: l ≡ x' :: l'
≡-:: refl refl = refl
```

from which one can deduce that two extensionally equal lists are equal:

```
==-≡ : {A : Set} {l l' : List A} → l == l' → l ≡ l'
==-≡ ==-[] = refl
==-≡ (==-:: x e) = ≡-:: x (==-≡ e)
```

Extensional equality on functions. Similarly again, we declare that two functions *f* and *g* of type $A \to B$ are *extensionally equal* when, for every element *x* of type *A*, we have $f(x) = g(x)$. Clearly, two equal functions are extensionally so

```
≡-ext : {A B : Set} → {f g : A → B} →
        f ≡ g → (x : A) → f x ≡ g x
≡-ext refl x = refl
```

(this function will be called `happly` in section 9.4.1), but the converse property cannot be shown because we have no induction principle at our disposal to show that two functions are equal. For instance, one cannot show

```
id-add-0 : (λ n → n + 0) ≡ (λ n → n)
```

Try it for yourself in order to get convinced.

This means that there is no proof of the following *function extensionality* principle:

```
FE : Set₁
FE = {A B : Set} {f g : A → B} → ((x : A) → f x ≡ g x) → f ≡ g
```

not to mention the *dependent function extensionality* principle, which is the generalization adapted to dependent functions:

```
DFE : Set₁
DFE = {A : Set} {B : A → Set} {f g : (x : A) → B x} →
     ((x : A) → f x ≡ g x) → f ≡ g
```

See [BPT17] for a proof of this fact.

This situation is deeply unsatisfactory: this means that we cannot really use equality to reason on functions. We could add (dependent) function extensionality as an axiom with

```
postulate funext : DFE
```

but we would not get very far because we would not have any computation rule associated to it, making proofs very hard in practice. Also, function extensionality seems to contradict the constructivity of proofs. Namely, a given function can be implemented in various ways, with various algorithms and various complexities (see section 2.1.1 for an example), and function extensionality seems to simply destroy it all, since we are considering all of them as equal. In fact, this reasoning would hold if proofs of equality did not have any content... but we will see that it is not the case in next section: the way we prove that two functions are equal is relevant here and cannot simply be discarded. A much better treatment of equality is proposed by homotopy type theory, which is presented in this chapter, and function extensionality will be a consequence of its main axiom, univalence, see section 9.4.9. It resolves the tension by considering that two equal things are not the same, but can be deformed one into the other.

9.1.6 Uniqueness of identity proofs. At some point in the 90s, people started to wonder: is there a proof-theoretic content in proofs of equality? Or more prosaically: can there be more than one proof of the equality between two given terms? This suggested investigating the provability of the following property called *uniqueness of identity proofs*

```
UIP : Set₁
UIP = {A : Set} {x y : A} (p q : x ≡ y) → p ≡ q
```

which states that two proofs of $x \equiv y$ for some terms x and y are necessarily equal.

In particular, in the case $x = y$, we know a particular proof of $x \equiv x$, namely refl(x). If we are only interested in such cases, we can also consider the following variant, which we call here *uniqueness of reflexivity proofs*:

```
URP : Set₁
URP = {A : Set} {x : A} (p : x ≡ x) → p ≡ refl
```

Clearly, URP is a particular case of UIP:

```
UIP-URP : UIP → URP
UIP-URP UIP r = UIP r refl
```

Interestingly, we can also recover UIP from URP. Namely, consider two identity proofs $p, q : x \equiv y$. We can picture the identities p and q as paths from x to y, as in the figure below:

$$x \overset{p}{\underset{q}{\frown}} y$$
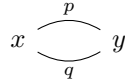
This diagram makes it plausible that showing that p is the same as q (in the sense that $p \equiv q$), should be equivalent to showing that the path from y to y, obtained as the concatenation of p taken backward followed by q is the same as the reflexivity on y. And indeed, one can show the following implication:

```
loop-≡ : {A : Set} {x y : A} (p q : x ≡ y) →
         trans (sym p) q ≡ refl → p ≡ q
loop-≡ refl q h = sym h
```

from which one deduces that URP implies UIP:

```
URP-UIP : URP → UIP
URP-UIP URP p q = loop-≡ p q (URP (trans (sym p) q))
```

The axiom K. A third equivalent property is called *K*, and is due to Streicher [Str93]. It can be thought of as the "Leibniz variant" (see section 9.1.4) of URP: if the only proof of an equality $x \equiv x$ is reflexivity then, in order to show that a property P depending on such a proof is valid, it should be enough to show it in the case of reflexivity. This property can thus be formulated as

```
K : Set₁
K = {A : Set} {x : A} →
    (P : (x ≡ x) → Set) → P refl → (p : x ≡ x) → P p
```

It is simple to show that URP implies K:

```
URP-K : URP → K
URP-K URP P r p = subst P (sym (URP p)) r
```

and that K implies URP:

```
K-URP : K → URP
K-URP K p = K (λ p → p ≡ refl) refl p
```

Note that K is a slight variant of the eliminator J, where we consider propositions depending on proofs of $x \equiv x$ (instead of $x \equiv y$), thus the name. However, K cannot be proved from J (try it!): this can be demonstrated by observing that the non-trivial models of homotopy type theory validate the latter but not the former: the first such model was found by Hofmann and Streicher [HS98], by interpreting types as groupoids.

Pattern matching without K. If we try to prove UIP or K in vanilla Agda, using pattern matching as usual, something unexpected happens, as first noticed by Coquand [Coq92b]: we succeed! Namely, we can show UIP by

```
UIP-proof : UIP
UIP-proof refl refl = refl
```

and K by

```
K-proof : K
K-proof P r refl = r
```

This means that Agda is not implementing dependent type theory exactly as we have presented it: in fact, the default pattern matching algorithm of Agda is simply too permissive. In order to use a saner algorithm, we should always start our files with

```
{-# OPTIONS --without-K #-}
```

and the above proofs of UIP and K will not be accepted anymore. The reason why the current pattern matching is enabled by default is that it simplifies proofs, if one is prepared to lose all information about identities: it can be shown that using this algorithm essentially amounts to adding UIP (and not more) to the dependent type theory [McB00].

9.2 Types as spaces

9.2.1 Intuition about the model. In order to better understand what logic looks like in the absence of uniqueness of identity proofs, one should be prepared to accept the following change of point of view: types should be interpreted not as booleans, nor as sets, but as *spaces*. Similarly, the elements of a type $A \to B$ should be interpreted not as implications, nor as functions, but as *continuous functions*. This interpretation is the starting point of *homotopy type theory* which was pioneered by Voevodsky and other people [Uni13]. In order to make this clear, even in Agda, starting from now, we will write `Type` instead of `Set` to designate the type of types, which can be done by defining

```
Type : (i : Level) → Set (lsuc i)
Type i = Set i
```

Types are not always sets!

Spaces. We will deliberately remain vague about what we mean by a space, but one can think of those as geometric shapes, in arbitrary dimension, i.e. as topological spaces, or as something that can be obtained by gluing segments, surfaces and volumes in arbitrary dimension. More details can be found in standard algebraic topology textbooks such as [Hat02]. Some examples in low dimensions are
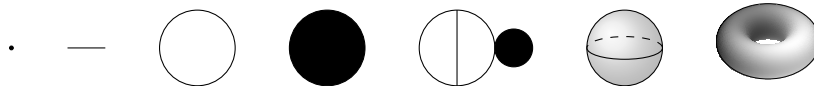
Most importantly, those spaces should be considered up to "deformations" which preserve the shape: we do not distinguish between spaces which look roughly the same.

Paths. The reason why this interpretation is useful to reason about identities is that we now have a representation for them: they correspond to paths, as we now explain. We write I for the *interval* space:

———

Concretely, it can be defined as the set $I = [0, 1]$ of reals between 0 and 1 (both included) equipped with the euclidean topology. A *path* in a space A from a point x to a point y is a continuous function

$$p : I \to A$$

such that $p(0) = x$ and $p(1) = y$:
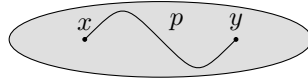
Such a path can also be thought of as a continuous way to go from x to y depending on a "time" parameter $t \in I$: at time $t = 0$ we are at x, at time $t = 1$ we are at y, and at a time t in between we are at $p(t)$.

Given a point x there is always a path from x to x, the *constant path*, which is the function defined as $p(t) = x$ for every $t \in I$. This corresponds to remaining at x.

Interpreting types. From now on, we are going to work with the following interpretation of types in mind:

- we interpret a type A as a space,

- an element x of type A will be seen as a point of A,

- an identity proof p in $\mathrm{Id}_A(x, y)$ as a path from x to y,

- a function $f : A \to B$ as a continuous function from A to B.

For this reason, we sometimes write $p : x \equiv y$ for a path from x to y. In particular, a reflexivity proof $\mathrm{refl}(x) : x \equiv x$ will be interpreted as the constant path from x to x. We insist on the fact that the interpretations of functions are always continuous, even if we omit mentioning it.

Given two elements $x, y : A$ and two identities $p, q : \mathrm{Id}_A(x, y)$, consider an identity $\alpha : \mathrm{Id}_{\mathrm{Id}_A(x,y)}(p, q)$ from p to q. Topologically, it will correspond to a continuous way of deforming the path p into the path q within paths from x to y, i.e. the endpoints are fixed. It thus corresponds to a surface:
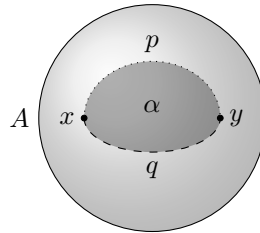
Similarly, paths between paths between paths correspond to volumes, and so on.

In particular, consider a type corresponding to the circle (should there be one). Given two distinct points x and y, we have two distinct paths p and q going from x to y:
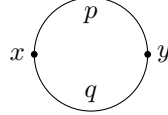
Moreover, since the circle is hollow, there can be no continuous way of deforming p into q. A type theory which can account for such a type will not validate the principle of uniqueness of identity proofs.

Homotopy equivalence. We mentioned that spaces are considered up to deformation and we now want to make more precise this notion of deformation we are using. We say that two continuous functions $f, g : A \to B$ are *homotopic* when for every point $x : A$ there is a path $f(x) \equiv g(x)$, which varies continuously with x. We write $f \sim g$ when f and g are homotopic.

Two spaces A and B are *homotopy equivalent* when there are two functions

$$f : A \to B \qquad \text{and} \qquad g : B \to A$$

such that

$$g \circ f \sim \text{id}_A \qquad \text{and} \qquad f \circ g \sim \text{id}_B$$

where $\text{id}_A : A \to A$ is the identity function, defined by $\text{id}_A(x) = x$ for x in A, and similarly for id_B. This is the notion we will use when we think of two spaces as being equivalent up to deformation: the adjective *homotopy* in homotopy type theory refers to the fact that we are considering spaces up to homotopy equivalence.

For instance, the space A reduced to a point x (on the left) is homotopy equivalent to the disk B (on the right):
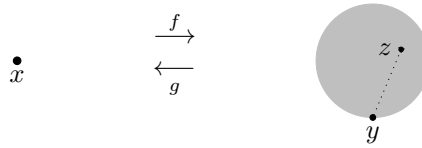
Namely, we can take the function $f : A \to B$ which takes x to some point y of B, and the function $g : B \to A$ which takes every point of B to x. This is a homotopy equivalence since, for the only point x of A we have $g \circ f(x) = x$, and for every point z of B there is a path from $y = f \circ g(z)$ to z, which depends continuously on z (see the picture).

Consider the variant of the preceding situation, where A is still reduced to a point x, but B is now a circle instead of a disk:
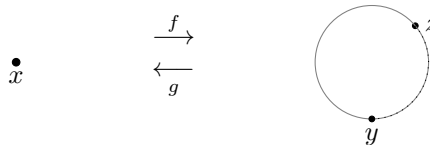
We can still define functions f and g in the same way. Moreover, given a point z in B there is still a path from $y = f \circ g(z)$ to z. However, there is no way of choosing such a path in a continuous way: when moving z around the circle, at some point the path has to jump from turning counterclockwise to clockwise, or from turning once around the circle to turning twice, etc. For this reason the point and the circle are not homotopy equivalent.

Remark 9.2.1.1. In the previous example, it can be noted that the circle has a hole whereas the point does not. It can be shown that homotopy equivalence preserves the number of holes in any dimension [Hat02] (these are called the Betti numbers and are closely related to homotopy groups), from which we could have easily seen that the two spaces are not equivalent. There is actually even a subtle converse to this property. A map $f : A \to B$ between spaces is a *weak homotopy equivalence* when it induces a bijection between the holes (in any dimension) of A and those of B (as a particular case, it should induce a bijection between the path components of A and those of B). When A and B are "nice" spaces, by which we mean gluing of disks (traditionally called *CW-complexes*), a map $f : A \to B$ is a weak homotopy equivalence if and only if it is a homotopy equivalence (this is known as the *Whitehead theorem*). The restriction to CW-complexes is not really a limitation here, because any space can be shown to be weakly homotopy equivalent to a CW-complex.

Eliminating identities. The elimination principle of identity types says that in order to show a property on a space containing a path p, it is enough to show this property on the corresponding space where p has been made a constant path. For instance, suppose that we want to show that the circle A satisfies UIP, i.e. any two paths are equal:
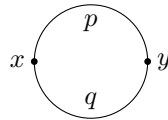
We thus begin our proof with

```
UIP : (x y : A) (p q : x ≡ y) → p ≡ q
UIP x y p q = ?
```

We begin by eliminating p, and Agda takes us to the proof

```
UIP : (x y : A) (p q : x ≡ y) → p ≡ q
UIP x .x refl q = ?
```

which corresponds to restricting to the space A'

obtained from the circle by assimilating p to a constant path. This is a perfectly valid thing to do because the spaces A and A' are clearly homotopy equivalent. However, if we proceed further and eliminate q, Agda gets us to

```
UIP : (x y : A) (p q : x ≡ y) → p ≡ q
UIP x .x refl refl = ?
```

which means that we are now restricting to the space A'' reduced to a point

$$\overset{x}{\bullet}$$

obtained from A' by assimilating q to the constant path. This step should not be valid because, as we have seen, A' and A'' are not homotopy equivalent. In fact, if we activate the flag `--without-K` of Agda, as we should always do, Agda rejects this last step by issuing an error:

```
I'm not sure if there should be a case for the constructor refl,
because I get stuck when trying to solve the following unification
problems (inferred index ≟ expected index):
  x₁ ≟ x₁
Possible reason why unification failed:
  Cannot eliminate reflexive equation x₁ = x₁ of type A₁ because K
  has been disabled.
when checking that the expression ? has type refl ≡ q
```

which is his verbose way of saying that you are trying to do something forbidden in the absence of axiom K.

Univalence. We will see that the type theory (without K) does not exactly match the intuition that we have of types as spaces: some properties that we expect to be shown cannot be proved. The reason is that we lack some ways of constructing equalities. For instance, we cannot construct non-trivial equalities between functions: in particular, we cannot prove function extensionality. In order for logic and topology to match precisely, one needs to assume an axiom, called *univalence*. It will be only be presented in section 9.4, but we will mention some of the properties which it allows to prove before that, in order to motivate the need for it (e.g. function extensionality will be a consequence of it).

9.2.2 The structure of paths. We shall now study the constructions and operations which are available on paths. The first one, which we have seen many times, is the construction of the constant path on a point x, which is simply given by `refl`. Given two paths $p : x \equiv y$ and $q : y \equiv z$ such that the end of p matches the beginning of q (both are y), we can build their *concatenation* $p \cdot q$, which is a path from x to z. If we see them as continuous functions $p : I \to A$ and $q : I \to A$, where I is the interval $[0, 1]$, this is defined as

$$(p \cdot q)(t) = \begin{cases} p(t) & \text{if } 0 \leqslant t \leqslant 1/2, \\ q(t - 1/2) & \text{if } 1/2 \leqslant t \leqslant 1. \end{cases}$$

In the following, we will generally not give such explicit constructions, and simply provide the formalization in Agda, which is in this case

```
_·_ : ∀ {i} {A : Type i} {x y z : A} →
      (p : x ≡ y) → (q : y ≡ z) → x ≡ z
refl · q = q
```

Of course, we have already seen this proof in section 6.6.2: this is simply the transitivity of $\equiv$. As expected, the constant path is a unit for concatenation on the left:

```
·-unit-l : ∀ {i} {A : Type i} {x y : A} →
           (p : x ≡ y) → refl · p ≡ p
·-unit-l p = refl
```

This does not mean that, given a path $p : x \equiv y$, the paths $\mathrm{refl} \cdot p$ and p are the same. In fact, they are not since the one on the left is

$$(\mathrm{refl} \cdot p)(t) = \begin{cases} t & \text{if } 0 \leqslant t \leqslant 1/2, \\ p(t - 1/2) & \text{if } 1/2 \leqslant t \leqslant 1. \end{cases}$$

and is a different function from p: we usually don't have $(\mathrm{refl} \cdot p)(t) = p(t)$ for every $t \in I$. They are however homotopic, in the sense that there is a path (i.e. a deformation) from the former to the latter (exercise: explicitly define this path). Similarly, the constant path is also a unit on the right for concatenation:

```
·-unit-r : ∀ {i} {A : Type i} {x y : A} →
           (p : x ≡ y) → p · refl ≡ p
·-unit-r refl = refl
```

and concatenation is associative:

```
·-assoc : ∀ {i} {A : Type i} {x y z w : A} →
          (p : x ≡ y) → (q : y ≡ z) → (r : z ≡ w) →
          (p · q) · r ≡ p · (q · r)
·-assoc refl refl refl = refl
```

Next, given a path $p : x \equiv y$, we can define the *inverse* path $p^{-1} : y \equiv x$ by $p^{-1}(t) = p(1 - t)$, i.e. the path p taken "backwards". In Agda, it is written `! p` (or `sym p`) and defined by

```
!_ : ∀ {i} {A : Type i} {x y : A} → x ≡ y → y ≡ x
! refl = refl
```

Again, we can show the expected properties such as the fact that it is a neutral element on the left:

```
·-inv-l : ∀ {i} {A : Type i} {x y : A} →
          (p : x ≡ y) → ! p · p ≡ refl
·-inv-l refl = refl
```

which means that taking a path backwards and then forward is the same (up to homotopy) as doing nothing (try it in the street). The same holds on the right:

```
·-inv-r : ∀ {i} {A : Type i} {x y : A} →
          (p : x ≡ y) → p · ! p ≡ refl
·-inv-r refl = refl
```

and taking the inverse twice does nothing:

```
!-! : ∀ {i} {A : Type i} {x y : A} → (p : x ≡ y) → ! (! p) ≡ p
!-! refl = refl
```

Groupoids. If we sum up the situation, given a type A, we have

- a set A of points,

- for every points x and y in A, we have a set $x \equiv y$ of paths from x to y,

- for every point x, we have a path $\mathrm{refl}(x) : x \equiv x$,

- for every points x, y, z and paths $p : x \equiv y$ and $q : y \equiv z$, we have a concatenation $p \cdot q : x \equiv z$,

such that

- concatenation is associative and admits constant paths as neutral elements on both sides,

- every path admits a path which is an inverse on both sides.

A *groupoid* is precisely this structure, if we assume that the two above axioms hold up to equality (as opposed to up to homotopy): it consists of a set (of points or *objects*), together with a set (of paths or *morphisms*) between any pair of points, equipped with a composition and identities (constant paths), such that composition is associative, unital and admits inverses. The first model of dependent type theory which did not validate UIP was actually constructed by interpreting types as groupoids [HS98]. It can be seen as a "degenerate" version of the model of spaces, in the sense that the only paths between paths are constant paths.

9.3 n-types

Now that we have this point of view on types as spaces, we can start classifying types depending on their topological properties. A particularly interesting classification is given by n-types, which are types which contain no holes of dimension $k > n$, for some natural number n.

9.3.1 Propositions. The most simple kind of types are *propositions* [Uni13, Section 3.3]. We can think of a proposition as either being

- a point meaning that it is *true*, or

- empty, meaning that it is *false*.

In particular, when it is true, we only allow for one point: if there were many, it would mean that there would be many reasons why the proposition would be true, which is not what we have in mind for propositions. One should be aware that the above description is slightly misleading:

- it will not be the case that we can prove that a proposition is either empty or not, i.e. either true or false, because we live in an intuitionistic world, where the excluded middle is not expected to hold,

- in true, we require that there is only one point, call it x_0, *up to homotopy*: this means that if there is another point x, it should be equal to x_0.

In both cases (true or false), we note that a proposition is such that any two points x and y are related by a path $x \equiv y$: this property holds by definition when the proposition is true and is vacuously true when the proposition is empty.

Definition. The previous remark suggests defining a predicate `isProp` by

```
isProp : ∀ {i} → Type i → Type i
isProp A = (x y : A) → x ≡ y
```

with the intended meaning that `isProp A` holds when the type A is a proposition.

Examples. We can show that $\bot$ is a proposition (it corresponds to the empty space):

```
⊥-isProp : isProp ⊥
⊥-isProp ()
```

or $\top$ is a proposition (it corresponds to the space with one point, namely `tt`):

```
⊤-isProp : isProp ⊤
⊤-isProp tt tt = refl
```

We can also show that the type of booleans is not a proposition since it has two points, `true` and `false`, which are not equal:

```
Bool-isn'tProp : ¬ (isProp Bool)
Bool-isn'tProp P with P true false
Bool-isn'tProp P | ()
```

We insist once more on the fact that types are handled up to homotopy, so that a disk

is also an acceptable proposition because it is homotopy equivalent to a point. However, there is a worrying situation with our definition: it seems that the circle C

should also be accepted by our definition although it is not equivalent to a point: after all, given any pair of points there is a path between them in the circle. However, C is not a proposition and one cannot show `isProp C`: the reason is that there is no way choosing such paths in a *continuous* way, and we are only allowed to manipulate continuous functions. Namely, one can convince himself that there is no way of choosing a path $p_{x,y} : x \equiv y$ for every pair of points x and y in C, in a way which is continuous in both x and y (the reasoning is similar to the one we have done above to show that the circle is not homotopy equivalent to a point).

The type of propositions. We can define the type of all propositions as

```
hProp : ∀ i → Type (lsuc i)
hProp i = Σ (Type i) isProp
```

(we call it `hProp` and not `Prop` because the latter is a reserved keyword in recent versions of Agda). It should be remarked that even though we know that this type should be small, because we add at most one point on `hProp` per type in `Type i`, the general rule for handling levels in Σ-types does not allow the result to be in `Type i`, because `Type i` is at level `i+1`, so that we have to assume that it forms a large type, i.e. at level `i+1` (this is further discussed in section 9.3.4, with the propositional resizing principle).

Operations on propositions. In the following, we will use the set-theoretic notations for usual operations on types and the logical notations for the corresponding operations on propositions:

on types	$\times$	$\sqcup$	$\rightarrow$	Π	Σ
on propositions	$\wedge$	$\vee$	$\Rightarrow$	$\forall$	$\exists$

(we are using here the more traditional notation $\sqcup$ instead of the usual Agda notation $\uplus$ for coproduct). The Curry-Howard correspondence allowed us to identify both lines, but now that we have a rich type theory, we can tear logic and types apart again! In order for this to make sense, we should check that the operations are well-defined on propositions, i.e. that the result is a proposition when applied to proposition. We will see that it is actually not always the case and that their definitions have to be adapted to properly operate on propositions.

Propositions are closed under products, i.e. the product of two propositions is itself a proposition:

```
×-isProp : ∀ {i j} {A : Type i} {B : Type j} →
           isProp A → isProp B → isProp (A × B)
×-isProp PA PB (a , b) (a' , b') with PA a a' , PB b b'
×-isProp PA PB (a , b) (.a , .b) | refl , refl = refl
```

We can therefore simply define the conjunction of propositions as their product:

```
_∧_ : ∀ {i j} → Type i → Type j → Type (lmax i j)
A ∧ B = A × B
```

Similarly, we expect that propositions are closed under function spaces, so that we can simply define implication as function space. In order to show this, it turns out that we have to assume function extensionality (which will become a theorem in section 9.4.9), because we have no useful way to show equalities between functions otherwise, see section 9.1.5. If this is assumed, one can show that $A \rightarrow B$ is a proposition as soon as B is:

```
→-isProp : ∀ {i j} {A : Type i} {B : Type j} →
           isProp B → isProp (A → B)
→-isProp PB f g = funext (λ x → PB (f x) (g x))
```

and similarly for Π-types, if we assume dependent function extensionality:

```
Π-isProp : ∀ {i j} {A : Type i} → {B : A → Type j} →
           ((x : A) → isProp (B x)) → isProp ((x : A) → (B x))
Π-isProp PB f g = funext (λ x → PB x (f x) (g x))
```

In particular, the negation $\neg A$ of a type A is always a proposition since it is the type $A \to \bot$ by definition, and $\bot$ is a proposition:

```
¬-isProp : ∀ {i} {A : Type i} → isProp (¬ A)
¬-isProp = →-isProp ⊥-isProp
```

The situation for the coproduct $A + B$ of two propositions is more delicate. We have to assume that the types A and B have an "empty intersection" in order to show that their coproduct is itself a proposition. Here, having an empty intersection amounts to supposing that $\neg(A \wedge B)$ holds, or equivalently that $A \Rightarrow B \Rightarrow \bot$ holds.

```
⊔-isProp : ∀ {i j} {A : Type i} {B : Type j} →
           isProp A → isProp B → (A → B → ⊥) → isProp (A ⊔ B)
⊔-isProp PA PB f (inl x) (inl y) = ap inl (PA x y)
⊔-isProp PA PB f (inl x) (inr y) = ⊥-elim (f x y)
⊔-isProp PA PB f (inr x) (inl y) = ⊥-elim (f y x)
⊔-isProp PA PB f (inr x) (inr y) = ap inr (PB x y)
```

(the operation `ap` above is another name for `cong`, see section 6.6.2). The condition on intersection is really needed. For instance, the type $\top$ is a proposition but the type $\top \sqcup \top$ is not because it has two points: one could show that it is not a proposition in the same way we were able to show that `Bool` was not a proposition in section 9.3.1 (after all, the types $\top \sqcup \top$ and `Bool` are isomorphic). An important consequence of the above lemma is that, for every proposition A, the type $\neg A \sqcup A$ is also a proposition:

```
isDec-isProp : ∀ {i} {A : Type i} → isProp A → isProp (isDec A)
isDec-isProp PA = ⊔-isProp ¬-isProp PA λ a' a → a' a
```

Above, `isDec A` is a simply a notation for `¬ A ⊔ A`, meaning that A is decidable: we have just shown that, for a proposition, being decidable is a proposition.

We will be able to give a proper definition of $\vee$ (not just disjoint propositions) in section 9.3.4, but we do not have the tools to do so for now. For similar reasons as for coproduct, propositions are not closed under Σ-types and we also defer the definition of the $\exists$ quantifier.

As a final remark about connectives on propositions, we mention that it would be cleaner and more conceptual to define them directly on `hProp`, i.e. have them provide the proof that they produce propositions. For instance, conjunction could be defined as

```
_∧_ : ∀ {i j} → hProp i → hProp j → hProp (lmax i j)
(A , PA) ∧ (B , PB) = (A × B) , ×-isProp PA PB
```

We choose not to do this here in order to keep closer to bare metal and avoid small lemmas which would obfuscate the code at first read.

Predicates and propositions. For any type A, the type `isProp A` is itself a proposition: being a proposition is a proposition. If this was not the case, there could be multiple reasons why a proposition could be a proposition, and the meaning of this would be rather obscure. The proof, which is called `isProp-isProp`, is deferred to section 9.3.2.

Up to now, we have formalized a *predicate* on a type `A` as a function `P` whose type is `A → Set`, see section 6.5.9. For the same reasons as above, such a function really deserves the name of predicate only when it is the case that `P x` is a proposition for every element `x` of type `A`. We can thus formalize the fact of being a predicate as

```
isPred : ∀ {i j} {A : Type i} → (A → Set j) → Set (lmax i j)
isPred {A = A} P = (x : A) → isProp (P x)
```

The function `isProp-isProp` described above shows that `isProp` is a predicate:

```
isProp-isPred : ∀ {j} → isPred {j = j} isProp
isProp-isPred A = isProp-isProp
```

Similarly, as expected, being a predicate is itself a predicate:

```
isPred-isPred : ∀ {i j} {A} → isPred (isPred {i} {j} {A})
isPred-isPred P = Π-isProp (λ x → isProp-isProp)
```

Propositional extensionality. On propositions, there are two sensible notions of being the same:

  – propositional equality ≡,

  – logical equivalence ⇔.

In Agda, the second one is defined as usual from implication an conjunction by

```
_↔_ : ∀ {i} → Type i → Type i → Type i
A ↔ B = (A → B) ∧ (B → A)
```

We now briefly investigate the relationship between the two.

It is easy to observe that two equal propositions are equivalent, by induction on their equality:

```
≡-to-↔ : ∀ {i} → {A B : Type i} → A ≡ B → A ↔ B
≡-to-↔ refl = (λ x → x) , (λ x → x)
```

The converse implication is called *propositional equality*, or *PE*:

$$(A \Leftrightarrow B) \Rightarrow (A \equiv B)$$

which can be defined in Agda by

```
PE : ∀ {i} → Type (lsuc i)
PE {i} = ∀ {A B : Type i} → isProp A → isProp B → A ↔ B → A ≡ B
```

This implication cannot be shown and could be added as an axiom if one wants to use it (for instance, we use it in order to show Diaconescu's theorem in section 9.3.4). In fact, we will add univalence, which is a generalization of propositional equality, as an axiom, and show that it implies propositional extensionality in section 9.4.10.

Propositions as ⊤ or ⊥. At the beginning of this section, we have indicated that a proposition should be either empty or a point (up to homotopy), i.e. either ⊥ or ⊤. But can we formalize this? A first idea would be to show, for any type A, the implication

$$\mathrm{isProp}(A) \Rightarrow (A \equiv \bot) \vee (A \equiv \top)$$

However, we will not be able to show this for any type A, because it would allow us to decide whether A is true or not, which we cannot because we live in an intuitionistic world. However, if we know that A holds then it should be equal to ⊤:

$$\mathrm{isProp}(A) \Rightarrow A \Rightarrow (A \equiv \top)$$

and if A does not hold then it should be equal to ⊥:

$$\mathrm{isProp}(A) \Rightarrow \neg A \Rightarrow (A \equiv \bot)$$

We currently cannot show that, but we will see in section 9.4.6 that it can be proved if we assume the univalence axiom.

Classical logic. If one is disposed to work with classical logic, as presented in section 2.5, one should add the law of excluded middle

```
LEM : ∀ {i} → Type (lsuc i)
LEM {i} = {A : Type i} → isProp A → A ∨ ¬ A
```

or double negation elimination

```
NNE : ∀ {i} → Type (lsuc i)
NNE {i} = {A : Type i} → isProp A → ¬ (¬ A) → A
```

as postulates. In the above formulation, the reader should note that we are restricting those laws to propositions: they are only intended to talk about logic. For instance, we expect that the law of excluded middle states that a proposition is true or not, not that we can decide whether any type is empty or not, and construct an element of this type in the latter case. This axiom is consistent with homotopy type theory [KL20]. However, the general form of the law of excluded middle

```
LEM' : ∀ {i} → Type (lsuc i)
LEM' {i} = {A : Type i} → A ∨ ¬ A
```

(not restricted to propositions) is inconsistent with the axiom of univalence: it not only implies that we can choose an element in every non-empty type, but also that we should be able in a continuous way, which is not possible, see section 9.4.7.

9.3.2 Sets. After having considered propositions, the next interesting kind of types are *sets* [Uni13, section 3.1]. Those are types which are collections of points (up to homotopy). A typical set is thus:
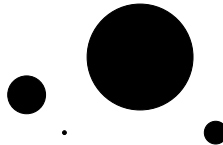
However, the circle

is not a set because it is not a collection of points. In a set, two points x and y are either in the same connected component, in which case they are equal in a unique way (up to homotopy), or they are in distinct components, in which case they are not equal. In other words, if they are equal, they should be uniquely so. This suggests defining the following predicate for sets:

```
isSet : ∀ {i} → Type i → Type i
isSet A = (x y : A) (p q : x ≡ y) → p ≡ q
```

Examples of sets. For instance, booleans form a set

```
Bool-isSet : isSet Bool
Bool-isSet false false refl refl = refl
Bool-isSet true  true  refl refl = refl
```

Natural numbers also form a set. First observe that, since equality is a congruence, every path $q : m \equiv n$ induces a path $p : m + 1 \equiv n + 1$:

```
suc-≡ : {m n : ℕ} → (m ≡ n) → (suc m ≡ suc n)
suc-≡ p = ap suc p
```

The path p is constructed from the path q by a direct application of `ap`, which is another name for `cong`. Now, we can show a lemma stating that every path $p : m + 1 \equiv n + 1$ is of this form, i.e. there are no more paths between successors than those induced by congruence:

```
suc-pred-≡ : {m n : ℕ} →
             (p : suc m ≡ suc n) → p ≡ ap suc (ap pred p)
suc-pred-≡ refl = refl
```

From there, we can show that $\mathbb{N}$ is a set, i.e. that any two paths $p, q : m \equiv n$ between natural numbers are equal. We proceed by induction on m and n. The base case where both are 0 is obvious, for the inductive case where both are successors, we can use the above lemma to reduce to the case where both p and q are obtained by congruence and we can use the induction hypothesis:

```
ℕ-isSet : isSet ℕ
ℕ-isSet zero zero refl refl = refl
ℕ-isSet (suc m) (suc n) p q =
  p                      ≡⟨ suc-pred-≡ p ⟩
  ap suc (ap pred p) ≡⟨ ap (ap suc) (ℕ-isSet m n _ _) ⟩
  ap suc (ap pred q) ≡⟨ sym (suc-pred-≡ q) ⟩
  q                      ∎
```

More generally, all the basic datatypes we usually use (natural numbers, strings, etc.) are sets. This includes the types from the previous section, since one can show that every proposition is a set, see below. Moreover, all usual type constructors (lists, vectors, etc.) preserve the fact of being a set.

Exercise 9.3.2.1. Show that the type `List A` is a set when `A` is a set.

Closure properties. Sets are closed under most usual operations (products, co-products, arrows, Π-types, Σ-types), as expected from set theory. As an illustration, let us show the closure under products. Recall from section 9.1.5 that, given two types A and B, a pair of paths $p : x \equiv x'$ in A and $q : y \equiv y'$ in B canonically induce a path from (x, y) to (x', y') in $A \times B$, that we abusively write $(p, q) : (x, y) \equiv (x', y')$ here:

```
×-≡ : ∀ {i j} {A : Type i} {B : Type j} {x x' : A} {y y' : B} →
      x ≡ x' → y ≡ y' → (x , y) ≡ (x' , y')
×-≡ refl refl = refl
```

Moreover, every path in $A \times B$ is equal to a path of this form. More precisely, a path $p : (x, y) \equiv (x', y')$ in $A \times B$ induces, by congruence under the projections, paths $p_A : x \equiv x'$ and $p_B : y \equiv y'$, and the path induced by p_A and p_B using previous function is equal to p, i.e. $p \equiv (p_A, p_B)$:

```
×-≡-η : ∀ {i} {j} {A : Type i} {B : Type j}
        {z z' : A × B} {p : z ≡ z'} →
        p ≡ ×-≡ (ap fst p) (ap snd p)
×-≡-η {p = refl} = refl
```

Finally, we can use this to show that the product $A \times B$ of the sets A and B is itself a set. Namely, given parallel paths p and q in $A \times B$, we have

$$p \equiv (p_A, p_B) \equiv (q_A, q_B) \equiv q$$

where the first and last equalities come from the previous observation, and the one in the middle follows from the fact that we have $p_A \equiv q_A$ and $p_B \equiv q_B$ because both A and B are sets:

```
×-isSet : ∀ {i j} {A : Type i} {B : Type j} →
          isSet A → isSet B → isSet (A × B)
×-isSet SA SB (x , y) (x' , y') p q =
  p ≡⟨ ×-≡-η ⟩
  ×-≡ (ap fst p) (ap snd p) ≡⟨ ap2 ×-≡
                                (SA x x' (ap fst p) (ap fst q))
                                (SB y y' (ap snd p) (ap snd q)) ⟩
  ×-≡ (ap fst q) (ap snd q) ≡⟨ sym ×-≡-η ⟩
  q ∎
```

Propositions are sets. Any proposition is a set [Uni13, Lemma 3.3.4]. This is intuitively expected because a proposition should be either empty or a point, and thus a particular case of a collection of points. Consider a proposition A, and two paths $p, q : x \equiv y$ between points x and y of A. In order to show that A is a set, we have to show that the paths p and q are equal, which is not easily done directly. Instead, we are going to show that both are equal to a third "canonical" path.
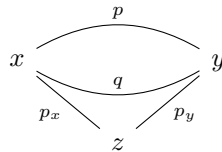
Fix a point z in A. Since A is a proposition, for every point x of A, there is a path $p_x : z \equiv x$. We now have a candidate for the canonical path: let's show that $p \equiv p_x^{-1} \cdot p_y$. By induction on p, this is immediate, since when $p = \text{refl}(x)$, we have $\text{refl}(x) \equiv p_x^{-1} \cdot p_x$, see section 9.2.2:

```
aProp-isSet-lem : ∀ {i} {A : Type i} {x y : A} → (P : isProp A) →
                  (z : A) (p : x ≡ y) → p ≡ ! (P z x) · (P z y)
aProp-isSet-lem {x = x} P z refl = sym (·-inv-l (P z x))
```

Similarly, we can show that $q \equiv p_x^{-1} \cdot p_y$, and therefore deduce $p \equiv q$, i.e. A is a set:

```
aProp-isSet : ∀ {i} {A : Type i} → isProp A → isSet A
aProp-isSet {A = A} P x y p q =
  (aProp-isSet-lem P x p) · (sym (aProp-isSet-lem P x q))
```

This result allows deducing the fact that being a proposition is itself a proposition using dependent function extensionality [Uni13, Lemma 3.3.5]. Namely, consider a type A and two proofs f, g that A is a proposition: those are functions taking two elements x and y of A and producing a path $x \equiv y$. By extensionality, is enough to show that we have $f\,x\,y \equiv g\,x\,y$ for every points $x, y : A$, which follows immediately from the fact that A is a proposition (by f or g).

```
isProp-isProp : ∀ {i} {A : Type i} → isProp (isProp A)
isProp-isProp f g =
  funext2 (λ x y → aProp-isSet f x y (f x y) (g x y))
```

Above, `funext2` is the obvious variant of `funext` for functions with two arguments.

Hedberg's theorem. An abstract reason why most usual types are sets is because they have decidable equality: *Hedberg's theorem* states that any type with a decidable equality is necessarily a set [Hed98, KECA16] and [Uni13, section 7.2]. For instance, we can decide the equality of natural numbers (see section 6.6.8), therefore they form a set (which we have already proved directly above).

We recall that a type A is said to be *decidable* when $\neg A \sqcup A$ holds, i.e. we can either show that it is empty or produce an element of it:

```
isDec : ∀ {i} (A : Type i) → Type i
isDec A = ¬ A ⊔ A
```

In particular, a type A has decidable equality when we can decide whether any two elements of A are equal or not:

```
isDecEq : ∀ {i} (A : Type i) → Type i
isDecEq A = (x y : A) → isDec (x ≡ y)
```

Although this is the property usually considered, it will turn out to be more convenient here to consider a variant of this property. We say that a type A has the property of *double negation elimination* if $\neg\neg A \to A$:

```
isNNE : ∀ {i} → Type i → Type i
isNNE A = ¬ (¬ A) → A
```

and we write `isNNEq` when its equality has this property:

```
isNNEq : ∀ {i} → Type i → Type i
isNNEq A = (x y : A) → isNNE (x ≡ y)
```

It is well known that decidability of a type implies double negation elimination:

```
isDec-isNNE : ∀ {i} {A : Type i} → isDec A → isNNE A
isDec-isNNE (inl a') a'' = ⊥-elim (a'' a')
isDec-isNNE (inr a)  _   = a
```

and therefore decidability of equality implies that equality has the double negation property. In this section, by "having a decidable equality", we will therefore without loss of generality mean "having an equality with the double negation elimination property".

Suppose that the type A has decidable equality. In order to show that A is a set, we have to show that any two paths $p, q : x \equiv y$ are equal. The proof strategy here is the same as above: we should show that p is equal to a "canonical" path of type $x \equiv y$, the path q will similarly be equal to this path and we will be able to conclude. The fact that A has decidable equality provides us with a canonical path between x and y. Namely, the existence of the path p implies that we have a proof $\lambda k.kp$ of $\neg\neg(x \equiv y)$ and the double negation elimination property provides us with a path $x \equiv y$:

```
nnePath : ∀ {i} {A : Type i} → isNNEq A →
          {x y : A} → (p : x ≡ y) → x ≡ y
nnePath N {x} {y} p = N x y (λ k → k p)
```

This path is canonical, in the sense that it does not depend on the choice of the path p. Namely, we know from section 9.3.1 that the type $\neg\neg(x \equiv y)$ is a proposition (any negation of a type is). In particular, given two paths p and q of type $x \equiv y$, the proofs $\lambda k.kp$ and $\lambda k.kq$ of $\neg\neg(x \equiv y)$ are equal and therefore induce equal paths of type $x \equiv y$ by elimination of double negation:

```
nnePathIndep : ∀ {i} {A : Type i} (N : isNNEq A) {x y : A}
               (p q : x ≡ y) → nnePath N p ≡ nnePath N q
nnePathIndep N {x} {y} p q =
  ap (N x y) ((¬-isProp (λ k → k p) (λ k → k q)))
```

In this way, we have constructed a canonical path $p_{x,y} : x \equiv y$, which depends only on x and y. Finally, we want to show that $p \equiv p_{x,y}$, i.e. the arbitrary path p is equal to the canonical one. By induction on p, this would require to show that $\mathrm{refl}(x) = p_{x,x}$, and there is no reason why this should hold. So instead, we consider a variant of the canonical path and show that $p \equiv p_{x,x}^{-1} \cdot p_{x,y}$. Namely, by induction on p, we are left proving $\mathrm{refl}(x) \equiv p_{x,x}^{-1} \cdot p_{x,x}$, which does hold, see section 9.2.2:

```
nnePathEq : ∀ {i} {A : Type i} (N : isNNEq A) {x y : A}
            (p : x ≡ y) → p ≡ ! (nnePath N refl) · nnePath N p
nnePathEq N {x} {y} refl = sym (·-inv-l (N x x (λ z → z refl)))
```

Finally, we can conclude that $p \equiv p_{x,x}^{-1} \cdot p_{x,y} \equiv q$ and therefore that A is a set:

```
Hedberg : ∀ {i} {A : Type i} (N : isNNEq A) → isSet A
Hedberg N x y p q =
  p ≡⟨ nnePathEq N p ⟩
  (! (nnePath N refl) · nnePath N p)
    ≡⟨ ap (λ nnp → ! (nnePath N refl) · nnp) (nnePathIndep N p q) ⟩
  (! (nnePath N refl) · nnePath N q)
    ≡⟨ sym (nnePathEq N q) ⟩
  q ∎
```

For instance, we have shown in section 6.6.8 that natural numbers have decidable equality. We thus have an alternative proof that they form a set by Hedberg theorem:

```
ℕ-isSet : isSet ℕ
ℕ-isSet = Hedberg (λ x y → isDec-isNNE (x ≟ y))
```

9.3.3 *n*-**types.** We now generalize the classification of types as propositions or sets into a full hierarchy of types.

Groupoids. It can be observed that the definition of being a set can be reformulated as:

```
isSet : ∀ {i} → Type i → Type i
isSet A = (x y : A) → isProp (x ≡ y)
```

i.e. a set is a type such that every pair of points x and y, the type $x \equiv y$ is a proposition. This reformulation suggests the next thing to try: we define a *groupoid* as a type such that for every pair of points x and y, the type $x \equiv y$ is a set:

```
isGroupoid : ∀ {i} → Type i → Type i
isGroupoid A = (x y : A) → isSet (x ≡ y)
```

In a groupoid, two points x and y might be equal in multiple ways, but there should be at most one equality between two paths $p, q : x \equiv y$. For instance, the circle (on the left) is a groupoid

but the sphere (on the right) is not a groupoid: between the point x and y there are two paths p and q and between those paths there are two non-homotopic paths (the deformations through the front or the back hemisphere).

The hierarchy. Continuing in this way, we define the notion of *n-type*, or a type of *homotopy level n*, by recursion on n [Uni13, Chapter 7]:

  – a 0-type is a set, and

  – an (*n*+1)-type is a type such that the type $x \equiv y$ is an *n*-type, for every points x and y.

In particular, a 1-type is a groupoid.

The intuition is that an n-type is a type which is trivial in dimensions higher than n, in the sense that it does not contain any non-trivial k-sphere for $k > n$. In low dimensions k, the k-spheres (or spheres in dimension k) can be pictured as follows:
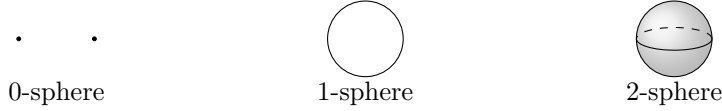
A 0-sphere thus consists of two points, a 1-sphere is a circle and a 2-sphere is a traditional sphere. For instance,

- a set (a 0-type) may contain two distinct points (a 0-sphere) but not a circle (a 1-sphere),

- a groupoid (a 1-type) may contain distinct points or circles but no 2-spheres,

and so on.

Negative types. The choice of $n = 0$ for sets is done in order to agree with traditional conventions in mathematics, but it can be extended a bit to negative numbers. We have seen that in a proposition is such that $x \equiv y$ is a 0-type (a set) for every pair of point x and y, so that it makes senses to define a (-1)-type as a proposition: if we adopt this convention, a 0-type is a type in which $x \equiv y$ is a (-1)-type, in accordance with the above definition.

Can we also make sense of a (-2)-type? In a (-1)-type, i.e. a proposition, for every pair of points x and y, we should have that $x \equiv y$ is a (-2)-type. Since in a proposition every pair of points is related by a unique path, a (-2)-type can be defined as a *contractible* type, i.e. a type which is a point up to homotopy, see below. If we go on with this reasoning, we find that a (-3)-type should still be a contractible type, so that we stop at dimension $n = -2$.

Contractible types. In Agda, the predicate of being contractible for a type can be defined as

```
isContr : ∀ {i} → Type i → Type i
isContr A = Σ A (λ x → (y : A) → x ≡ y)
```

It expresses the fact that a type is contractible when it contains a point x such that for every point y there is a path p_y from x to y. For instance, the type $\top$ is contractible since every point of it is equal to the only constructor `tt`:

```
⊤-isContr : isContr ⊤
⊤-isContr = tt , (λ { tt → refl })
```

Once again, it might seem that the circle is contractible because there is a path between any two pair of points, but it is not so because the choice of the path p_y has to be made continuously in y, which is not possible for the circle. A contractible type is thus homotopy equivalent to a point:

Apart from ⊤, an interesting contractible type is the *singleton* at a point x in a type A, which consists of all the points of A equal to x:

```
Singleton : ∀ {i} {A : Type i} → A → Type i
Singleton {A = A} x = Σ A (λ y → x ≡ y)
```

Such a type is always contractible:

```
Singleton-isContr : ∀ {i} {A : Type i} (x : A) →
                    isContr (Σ A (λ y → x ≡ y))
Singleton-isContr x = (x , refl) , λ { (y , refl) → refl }
```

Since a contractible type contains only one point up to homotopy, all its elements are necessarily equal, i.e. a contractible type is a proposition:

```
Contr-isProp : ∀ {i} {A : Type i} → isContr A → isProp A
Contr-isProp (x , p) y z with p y | p z
... | refl | refl = refl
```

(we will generalize this below when showing the cumulativity property).

n-types in Agda. We can define a predicate hasLevel such that hasLevel $(n+2)$ A holds when A is an n-type (we start at $n = -2$ instead of $n = 0$) by

```
hasLevel : ∀ {i} → ℕ → Type i → Type i
hasLevel zero    A = isContr A
hasLevel (suc n) A = (x y : A) → hasLevel n (x ≡ y)
```

Remark 9.3.3.1. Note that for a type A, being a (-1)-type according to the above definition (i.e. satisfying hasLevel 1) requires slightly more than the previous definition of propositions: for every pair of points x and y, there should be a path $p : x \equiv y$ as before, but we should also show that for every other path $q : x \equiv y$, we have $p \equiv q$. However, the second requirement is automatic if we carefully chose paths so that the two definitions coincide:

```
isProp-is1Type : ∀ {i} → {A : Type i} → isProp A → hasLevel 1 A
isProp-is1Type p x y = ! (p x x) · p x y ,
                       λ { refl → ·-inv-l (p x x) }
```

(we are using the same trick here than for Hedberg's theorem, see section 9.3.2).

Cumulativity. We have seen in section 9.3.2 that a proposition is a set. More generally, following the same ideas, one can show that every n-type is an $(n + 1)$-type. This entails that the hierarchy of n-types is *cumulative* in the sense that an n-type is an m-type for every $n \leqslant m$. This is shown by induction on n. For the base case, we have to show that a contractible type A (i.e. a (-2)-type) is also a proposition (i.e. a (-1)-type). Since A is contractible there is a point a in A and a path $p_x : a \equiv x$ for every point x in A. In order to show that A is a proposition, we have to show that, for every points x and y in A, we have a path $x \equiv y$: we can simply take $p_x^{-1} \cdot p_y$ (and every other path $q : x \equiv y$ is easily shown to be equal to this one by induction on q). The inductive case is simple. Formally,

```
hasLevel-cumul : ∀ {i} {n : ℕ} {A : Type i} →
                    hasLevel n A → hasLevel (suc n) A
hasLevel-cumul {_} {zero} (a , p) x y =
                    ! (p x) · p y , λ { refl → ·-inv-l (p x) }
hasLevel-cumul {_} {suc n} L x y = hasLevel-cumul (L x y)
```

The property of being an n-type. One can show that the property of being an n-type is a proposition: a type either is an n-type or not, but there cannot be multiple ways in which a type is an n-type.

For the base case, one has to show that being contractible is a proposition. Suppose given two proofs (x, p) and (y, q) that a type A is contractible, where x (resp. y) is a point of A and p (resp. q) associates to every point z of A a path $x \equiv z$ (resp. $y \equiv z$). Showing that these two proofs are equal amounts to showing that $x \equiv y$, which is given by p_y, and that $p \equiv q$. Assuming function extensionality, this last point is equivalent to showing that, for every point z in A, the paths $p_z : x \equiv z$ and $q_z : x \equiv z$ are equal, up to some transport of the first. Since A is contractible (we have a proof (x, p) of it), it is a 0-type (i.e. a set) by cumulativity, and therefore any two parallel paths in it are equal, thus $p_z \equiv q_z$:

```
isContr-isProp : ∀ {i} {A : Type i} → isProp (isContr A)
isContr-isProp {_} {A} (x , p) (y , q) =
  Σ-≡ (p y) (funext (λ z → fst (A-isSet y z _ (q z))))
    where
    A-isSet : hasLevel 2 A
    A-isSet = hasLevel-cumul (hasLevel-cumul (x , p))
```

The inductive case is handled immediately using function extensionality:

```
hasLevel-isProp : ∀ {i} {A : Type i}
                  (n : ℕ) → isProp (hasLevel n A)
hasLevel-isProp zero = isContr-isProp
hasLevel-isProp (suc n) f g =
  funext2 (λ x y → hasLevel-isProp n (f x y) (g x y))
```

9.3.4 Propositional truncation. We would now like to construct an operation, called propositional truncation, which turns an arbitrary type A into a proposition $\|A\|$, as detailed in [Uni13, section 3.7]. The intuition is that if a term of type A is a particular proof that A holds, a term of type $\|A\|$ is a witness that there exists a proof for A, but does not contain the information of an actual proof. Therefore, the type $\|A\|$ should be empty when A is and a point otherwise. If A is decidable, this operation is easy to define: either A or $\neg A$ holds, and we respectively define $\|A\| = \top$ or $\|A\| = \bot$. However, since we do not live in a classical world, we cannot define propositional truncation in this way. A more faithful description is that the propositional truncation starts from the type A and adds a path between any pair of points in order to turn it into a proposition, see section 9.5.4.

Rules. Propositional truncation is not a definable operation and has to be added as a new construction to the logic. We extend the syntax of expressions by

$$e ::= \dots \mid \|e\| \mid \|e\|_{\text{isProp}} \mid |e| \mid \text{rec}(e, e', x \mapsto e'')$$

where

- $\|A\|$ is the propositional truncation of A,

- $\|A\|_{\text{isProp}}$ is a proof that $\|A\|$ is a proposition, and

- $|t|$ provides a proof that $\|A\|$ is non-empty when there is a term t of type A,

- $\text{rec}(t, B, x \mapsto u)$ is the eliminator for truncated types.

The formation rules state that the propositional truncation $\|A\|$ exists for every type A and is a proposition:

$$\frac{\Gamma \vdash A : \text{Type}}{\Gamma \vdash \|A\| : \text{Type}} \; (\|\|_{\text{F}}) \qquad\qquad \frac{\Gamma \vdash A : \text{Type}}{\Gamma \vdash \|A\|_{\text{isProp}} : \text{isProp}(A)} \; (\|\|'_{\text{F}})$$

The introduction rule states that the propositional truncation $\|A\|$ is non-empty when A is

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash |t| : \|A\|} \; (\|\|_{\text{I}})$$

The elimination rule states that if we have an element of $\|A\|$, then we can assume that we have an element of A provided that the type we are currently proving (or "eliminating into") is a proposition:

$$\frac{\Gamma \vdash t : \|A\| \qquad \Gamma, x : A \vdash u : B \qquad \Gamma \vdash P : \text{isProp}(B)}{\Gamma \vdash \text{rec}(t, B, x \mapsto u) : B} \; (\|\|_{\text{E}})$$

The computation rule states that the element of A given by the elimination rule above is t when the witness given for $\|A\|$ is $|t|$:

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A \vdash u : B \qquad \Gamma \vdash P : \text{isProp}(B)}{\Gamma \vdash \text{rec}(|t|, B, x \mapsto u) = u[t/x] : B} \; (\|\|_{\text{C}})$$

The uniqueness rule is

$$\frac{\Gamma \vdash t : \|A\| \qquad \Gamma \vdash P : \text{isProp}(A)}{\Gamma \vdash |\text{rec}(t, A, x \mapsto x)| = t : \|A\|} \; (\|\|_{\text{U}})$$

Remark 9.3.4.1. For simplicity, we have given the rules in the non-dependent case, which is the most useful one in practice. For full generality, we should allow B to depend on $\|A\|$ and adapt the rules accordingly. For instance, the elimination rule should be

$$\frac{\Gamma, x : \|A\| \vdash B \qquad \Gamma, x : A \vdash u : B[|x|/x] \qquad \begin{array}{c} \Gamma \vdash t : \|A\| \\ \Gamma, x : \|A\| \vdash P : \text{isProp}(B) \end{array}}{\Gamma \vdash \text{rec}(t, x \mapsto B, x \mapsto u) : B[t/x]} \; (\|\|_{\text{E}})$$

Definition. This construction can be implemented in Agda, by postulating axioms corresponding to the rules. Formation is

```
postulate ‖_‖ : ∀ {i} → Type i → Type i
postulate ‖‖-isProp : ∀ {i} {A : Type i} → isProp ‖ A ‖
```

introduction is

```
postulate |_| : ∀ {i} {A : Type i} → A → ∥ A ∥
```

elimination is

```
postulate ∥∥-rec : ∀ {i j} {A : Type i} {B : Type j} →
                   isProp B → (A → B) → (∥ A ∥ → B)
```

computation is

```
postulate ∥∥-comp : ∀ {i j} {A : Type i} {B : Type j} →
                    (P : isProp B) (f : A → B) (x : A) →
                    ∥∥-rec P f | x | ≡ f x
```

and uniqueness is

```
postulate ∥∥-eta : ∀ {i} {A : Type i} (P : isProp A) (x : ∥ A ∥) →
                   | ∥∥-rec P id x | ≡ x
```

Logical connectives. Remember from section 9.3.1 that we had difficulties defining the disjunction of propositions because the coproduct of two propositions is not a proposition in general (we can only show that it is a set). Now that we have the propositional truncation at hand, we can use it on order to squash the result of the coproduct into a proposition. We can thus define disjunction as

```
_∨_ : ∀ {i j} → Type i → Type j → Type (lmax i j)
A ∨ B = ∥ A ⊔ B ∥
```

The disjunction of two propositions is now a proposition by definition. Similarly, the existential quantification is a truncated variant of Σ-types:

```
∃ : ∀ {i j} → (A : Type i) → (A → Type j) → Type (lmax i j)
∃ A B = ∥ Σ A B ∥
```

The axiom of choice. In order to illustrate the difference between operations and their truncated variants, let us consider the possible implementations of the axiom of choice in type theory, see [Uni13, section 3.8]. Recall from section 5.3.2 that, in set theory, a possible formulation of this axiom states that, given a relation $R \subseteq A \times B$ between sets A and B such that every element x of A is in relation with at least one element y of B contains a function. In type theory, the naive translation of this is the formula

```
CAC : ∀ {i j k} → Type (lmax (lmax (lsuc i) (lsuc j)) (lsuc k))
CAC {i} {j} {k} = {A : Type i} {B : Type j}
    (R : A → B → Type k) →
    (r : (x : A) → Σ B (λ y → R x y)) →
    Σ (A → B) (λ f → (x : A) → R x (f x))
```

is called the *constructive axiom of choice*, or *CAC*, and we have seen in section 6.5.8 that this formula is easily proved. Namely, the argument of type

$$(x : A) → Σ B (λ y → R x y)$$

witnesses the fact that every element of A is in relation with some element of B. A term of this type is a function r which to every $x \in A$ associates a pair consisting of an element $y \in B$ together with a proof that the pair (x, y) is in the relation R. From this data it is easy to construct a function $A \to B$ (by post-composing r with the first projection) and a proof that we have $(x, r(x))$ in the relation R for every $x \in A$:

```
cac : ∀ {i j k} → CAC {i} {j} {k}
cac R f = (λ x → fst (f x)) , (λ x → snd (f x))
```

In some sense this was "too easy", because the function r directly provided us with a way to construct a suitable element of B from an element of A.

A more faithful way of implementing the axiom of choice in type theory consists, instead of supposing that we have a function r as above, in only supposing the existence of such a function, i.e. that its propositional truncation is inhabited, i.e. we use an existential quantification instead of a Σ-type. Similarly, as a result, we only want to show that there exists a suitable function from A to B, without explicitly constructing it. The "right" formulation of the axiom of choice is thus:

```
AC : ∀ {i j k} → Type (lmax (lmax (lsuc i) (lsuc j)) (lsuc k))
AC {i} {j} {k} = {A : Type i} {B : Type j} →
  isSet A → isSet B →
  (R : A → B → Type k) →
  ((x : A) (y : B) → isProp (R x y)) →
  (r : (x : A) → ∃ B (λ y → R x y)) →
  ∃ (A → B) (λ f → (x : A) → R x (f x))
```

Note that, since we are serious about homotopy levels, we have also restricted to the case where A and B are sets and $R\,x\,y$ is a proposition for every element x of A and y of B (the axiom without this restriction would be inconsistent with univalence [Uni13, Lemma 3.8.5]). There is also a dependent variant of this axiom (where the type B is allowed to depend on A):

```
DAC : ∀ {i j k} → Type (lmax (lmax (lsuc i) (lsuc j)) (lsuc k))
DAC {i} {j} {k} = {A : Type i} {B : A → Type j} →
  isSet A → ((x : A) → isSet (B x)) →
  (R : (x : A) → B x → Type k) →
  ((x : A) (y : B x) → isProp (R x y)) →
  (r : (x : A) → ∃ (B x) (λ y → R x y)) →
  ∃ ((x : A) → B x) (λ f → (x : A) → R x (f x))
```

It can be shown that AC and DAC are equivalent (exercise: show it). Finally, these axioms are also equivalent to the following axiom

```
PAC : ∀ {i j} → Type (lmax (lsuc i) (lsuc j))
PAC {i} {j} = {A : Type i} {B : A → Type j} →
  isSet A → ((x : A) → isSet (B x)) →
  ((x : A) → ‖ B x ‖) → ‖ ((x : A) → B x) ‖
```

which is close to the usual alternative formulation of the axiom of choice: a product of non-empty sets is non-empty, see section 5.3.2.

Diaconescu. We are now in position of formally proving *Diaconescu's theorem*, which states that

<div align="center">the axiom of choice implies the excluded middle.</div>

The traditional proof of this theorem was presented in section 5.3.3 and the reader is advised to read again the proof there before going on with current section, which we learned from [Alt19]. We suppose here that both function extensionality (see section 9.1.5) and propositional extensionality (see section 9.3.1) hold in this section, both being consequences of univalence.

We take for granted that the following formulation of the axiom of choice holds

```
PAC : ∀ {i j} → Type (lmax (lsuc i) (lsuc j))
PAC {i} {j} = {A : Type i} {B : A → Type j} →
  ((x : A) → ‖ B x ‖) → ‖ ((x : A) → B x) ‖
```

It can be remarked that we are not very serious about homotopy levels, i.e. we do not restrict to the case where A and the B x are supposed to be sets: adding this does not bring any interesting difficulty, but makes the proofs a bit longer and thus more difficult to read. We suppose fixed an arbitrary proposition P in Type i for some level i (here also, P should be taken to be a proposition if we were more rigorous) and our goal is to show

<div align="center">P ∨ ¬ P</div>

We write U for the set of non-empty subsets of booleans:

```
U = Σ (Bool → Type i) (λ Q → ∃ Bool Q)
```

An element of U consists of a subset Q of the booleans, encoded here as a predicate on booleans ($Q\,b$ holds when a boolean b belongs to the set) together with a proof that the set is non-empty ($Q\,b$ holds for some boolean b). In particular, this set contains two elements of interest for us: the set

$$F = \{b \in \text{Bool} \mid b = 0 \vee P\}$$

which is non-empty because it contains 0, formalized as

```
F : U
F = (λ b → b ≡ false ∨ P) , | false , | inl refl | |
```

and the set

$$T = \{b \in \text{Bool} \mid b = 1 \vee P\}$$

which is non-empty because it contains 1, formalized as

```
T : U
T = ((λ b → b ≡ true ∨ P)) , | true , | inl refl | |
```

An element Q of U consists of a subset Q' of Bool together with a proof Q'' that Q' is non-empty. The family consisting of all Q' such that Q belongs to U is thus a family of non-empty sets and, by the axiom of choice, it is non-empty: we have a function f which to every element Q of U associates an element of Q'. We will prove in the function

```
dec : ((Q : U) → Σ Bool (fst Q)) → P ∨ ¬ P
```

that this entails that $P \vee \neg P$ holds, from which we will be able to conclude as explained above:

```
Diaconescu : isProp P → PAC → P ∨ ¬ P
Diaconescu prop ac = ‖‖-rec ‖‖-isProp dec
  (ac {A = U} {B = (λ Q → Σ Bool (fst Q))} (λ Q → snd Q))
```

The crux of this proof is thus the function `dec`. It proceeds by case analysis on $f\,F$ and $f\,T$:

> – if $f\,F$ is true then $\mathsf{true} \equiv \mathsf{false} \vee P$ holds and thus P holds,
>
> – if $f\,T$ is false then $\mathsf{false} \equiv \mathsf{true} \vee P$ holds and thus P holds,
>
> – if $f\,F$ is false and $f\,T$ is true then we can show that $\neg P$ holds.

The subtle case is the last one, when $f\,F$ is false and $f\,T$ is true, because this entails that $\mathsf{false} \equiv \mathsf{false} \vee P$ and $\mathsf{true} \equiv \mathsf{true} \vee P$ hold, from which we cannot extract information. However, we can show that $\neg P$ holds in this case. Namely, suppose that P holds (we write x for its proof) and let us deduce $\bot$. Since P holds, by definition of F and T we have $F\,b \Leftrightarrow T\,b$ for every boolean b, thus $F\,b \equiv T\,b$ by propositional extensionality, and thus $F \equiv T$ by function extensionality:

```
F≡T : F ≡ T
F≡T =
  Σ-≡
    (funext
      λ {
        false → propext ‖‖-isProp ‖‖-isProp
                   ((λ _ → right x) , (λ _ → right x)) ;
        true  → propext ‖‖-isProp ‖‖-isProp
                   ((λ _ → right x) , (λ _ → right x))
      })
    (‖‖-isProp (transport (∃ Bool) _ (snd F)) (snd T))
```

From there, we can deduce that the boolean of $f\,F$ is equal to the boolean of $f\,T$ (recall that $f\,Q$ is a pair consisting of a boolean and a proof that it belongs to Q):

```
fF≡fT : fst (f F) ≡ fst (f T)
fF≡fT = ap (λ Q → fst (f Q)) F≡T
```

However, we know that those booleans are respectively `false` and `true`, and we can deduce that $\mathsf{false} \equiv \mathsf{true}$

```
absurd : P → (fst (f F) ≡ false) → (fst (f T) ≡ true) →
         false ≡ true
absurd x ff ft = transport2 _≡_ ff ft fF≡fT
```

from which we conclude to an absurdity. The proof of `dec` is finally

```
dec : ((Q : U) → Σ Bool (fst Q)) → P ∨ ¬ P
dec f with inspect (f F) | inspect (f T)
dec f | (true  , p) , _ | (true  , q) , _ =
  ‖‖-rec ‖‖-isProp (⊔-elim (λ ()) (λ x → | inl x |)) p
dec f | (true  , p) , _ | (false , q) , _ =
  ‖‖-rec ‖‖-isProp (⊔-elim (λ ()) (λ x → | inl x |)) p
dec f | (false , p) , _ | (false , q) , _ =
  ‖‖-rec ‖‖-isProp (⊔-elim (λ ()) (λ x → | inl x |)) q
dec f | (false , p) , k | (true  , q) , l =
  | inr (λ x → case absurd x (ap fst k) (ap fst l) of λ ()) |
```

Note that we don't directly match f F because we would lose the fact that the result of the match is equal to f F (and similarly for f T). Instead, we use inspect, which is defined by

```
inspect : ∀ {i} {A : Type i} (x : A) → Σ A (λ y → x ≡ y)
inspect x = x , refl
```

and allows retrieving both the result of the match and the equality with the matched value (using the terminology from section 9.3.3, this function returns an element of the singleton at x).

Revealing truncation. As explained above, propositional truncation erases proofs, keeping only the existence of a proof. However, sometimes knowing the existence of a witness is enough to reconstruct this witness [Esc19]. For instance, suppose that we are given a function $f : \mathbb{N} \to \mathbb{N}$ and we know that this function admits a root (i.e. a number n such that $f(n) = 0$), then we can actually construct root of f: we compute $f(0)$, $f(1)$, $f(2)$, and so on, until we find a natural number n such that $f(n) = 0$. The point is that knowing the existence of the root ensures that this process will eventually terminate. This can be formalized and we are going to prove

$$(\exists (n : \mathbb{N}).f(n) = 0) \Rightarrow (\Sigma(n : \mathbb{N}).f(n) = 0)$$

or, unfolding the notations,

$$\|\Sigma(n : \mathbb{N}).f(n) = 0\| \Rightarrow \Sigma(n : \mathbb{N}).f(n) = 0$$

We are thus able to extract a witness from knowing its existence. Note that the fact that $\mathbb{N}$ can be enumerated is crucial here: the implication $\|A\| \Rightarrow A$ does not hold in general, for an arbitrary type A. For instance, if f was of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, we would not expect to be able to construct a root from knowing its existence, because the type of functions $\mathbb{N} \to \mathbb{N}$ is not countable.

So, suppose that we have a proof E of $\exists (n : \mathbb{N}).f(n) = 0$ and we want to prove the proposition R which is $\Sigma(n : \mathbb{N}).f(n) = 0$. We cannot directly provide the required natural number n (we cannot magically guess the root) and we cannot use the hypothesis E: in order to do so, we would have to use the eliminator for propositional truncation, which we cannot do because the goal we are proving is not a proposition. Namely, the type R is a set, the set of all roots of f, and not a proposition (f might admit multiple roots). However, we can take a variant of this type in order to have a proposition: instead of constructing any root of f, we are going to construct a particular one, say the

smallest one. Namely, the set R' of natural numbers which is a smallest root of f contains exactly one element (the smallest root of f) and will thus be a proposition. We can prove it by using elimination of propositional truncation on E and then conclude that we have an element of R because we have the implication $R' \Rightarrow R$ (a smallest root of f is a root).

In Agda, we are going to reason on an arbitrary predicate P on natural numbers, our above example being the particular case where $P\,n$ is $f(n) = 0$. We can define a predicate isFirst such that a natural number n satisfies isFirst n when n is the smallest natural number for which P holds:

```
isFirst : ∀ {i} (P : ℕ → Type i) → ℕ → Type i
isFirst P n = P n × ((m : ℕ) → P m → n ≤ m)
```

Moreover, using antisymmetry of the order on natural numbers, two smallest numbers satisfying a property are equal (in other words, the smallest natural number to satisfy a property is unique, when it exists):

```
isFirst-≡ : ∀ {i} (P : ℕ → Type i) → {m n : ℕ} →
            isFirst P m → isFirst P n → m ≡ n
isFirst-≡ P {m} {n} (Pm , Fm) (Pn , Fn) =
  ≤-antisym (Fm n Pn) (Fn m Pm)
```

Using this, and the closure of properties of propositions under conjunction and Π-types, we can show that if P is a predicate on natural numbers, in the sense that $P\,n$ is a proposition for every natural number n, then the type of first natural numbers to satisfy this predicate is a proposition:

```
first-isProp : ∀ {i} (P : ℕ → Type i) → ((n : ℕ) → isProp (P n)) →
               isProp (Σ ℕ (isFirst P))
first-isProp P prop =
  Σ-isProp
    (λ n → ∧-isProp
            (prop n)
            (Π-isProp (λ n → Π-isProp (λ Pn → ≤-isProp))))
    (λ m n → isFirst-≡ P)
```

Next, our goal is to show that if we know an arbitrary natural number m satisfying a predicate P then we can construct the smallest one. In order to perform inductions, it will be useful to consider the type of the smallest natural number greater than a fixed number k satisfying a proposition:

```
isFirst-from : ∀ {i} → ℕ → (P : ℕ → Type i) → ℕ → Type i
isFirst-from k P n = isFirst (λ n → k ≤ n × P n) n
```

We will also use the following "downward" induction principle, which states that if we know that $P\,m$ holds and $P\,(n+1)$ implies $P\,n$ for an arbitrary number n, then $P\,n$ holds for every $n \leqslant m$. Formally, it can be expressed as

```
rec-down : ∀ {i} (P : ℕ → Type i) (m : ℕ) →
           P m → ((n : ℕ) → n < m → P (suc n) → P n) →
           (n : ℕ) → n ≤ m → P n
```

and its proof is left as an exercise to the reader. Suppose given a decidable predicate P (i.e. $P\,n \vee \neg(P\,n)$ holds for every n), for which we know a number m

such that $P\,m$ holds. By downward induction on $k \leqslant m$, we can construct the smallest number greater than k satisfying P. For the inductive step, if we know the smallest one n greater than $k + 1$ then the smallest one greater than k is k if $P\,k$ is satisfied or n otherwise (we need to be able to decide $P\,k$ to be able to perform this case analysis). Formally,

```
find-first-from : ∀ {i} (P : ℕ → Type i) →
                    ((n : ℕ) → isDec (P n)) →
                    (m : ℕ) → P m →
                    (k : ℕ) → k ≤ m → Σ ℕ (λ n → isFirst-from k P n)
find-first-from P dec m Pm k k≤m =
  rec-down
    (λ k → Σ ℕ (λ n → isFirst-from k P n))
    m
    (m , (≤-refl , Pm) , (λ { n (m≤n , Pn) → m≤n }))
    ind
    k k≤m
  where
    ind : (k : ℕ) → k < m →
          Σ ℕ (λ n → isFirst-from (suc k) P n) →
          Σ ℕ (λ n → isFirst-from k P n)
    ind k k<m (n , Pn) with dec k
    ind k k<m (n , (k+1≤n , Pn) , Fn) | inl ¬Pk =
      n , (≤-trans (n≤1+n k) k+1≤n , Pn) ,
      λ { i (k≤i , Pi) →
        case split-≤ k≤i of λ {
          (inl k≡i) → ⊥-elim (¬Pk (transport P (sym k≡i) Pi)) ;
          (inr k<i) → Fn i (k<i , Pi)
        }
      }
    ind k k<m _ | inr Pk =
      k , (≤-refl , Pk) , λ { n (k≤n , Pn) → k≤n }
```

where `split-≤` is

```
split-≤ : {m n : ℕ} → m ≤ n → m ≡ n ⊔ m < n
```

(proof left to the reader). We can thus construct the first natural number n satisfying P, by applying previous lemma to the case $k = 0$:

```
find-first : ∀ {i} (P : ℕ → Type i) → ((n : ℕ) → isDec (P n)) →
             (m : ℕ) → P m → Σ ℕ (λ n → isFirst P n)
find-first P dec m Pm with find-first-from P dec m Pm 0 z≤n
find-first P dec m Pm | n , (_ , Pn) , Fn =
  n , (Pn , λ n Pn → Fn n (z≤n , Pn))
```

It is now time to return to our original problem. Given a function $f : \mathbb{N} \to \mathbb{N}$ for which we have a proof E that $\exists(n : \mathbb{N}).f(n) = 0$. We can use the elimination principle for propositional truncation in order to show $\Sigma(n : \mathbb{N}).\mathrm{isFirst}(f(n) = 0)$, which is a proposition, and we are left with showing

$$(\Sigma(n : \mathbb{N}).f(n) = 0) \Rightarrow \Sigma(n : \mathbb{N}).\mathrm{isFirst}(f(n) = 0)$$

i.e. knowing a root of f we have to construct the smallest one, which is precisely the purpose of our `find-first` function above:

```
extract-first-root : (f : ℕ → ℕ) →
                     ∃ ℕ (λ n → f n ≡ zero) →
                     Σ ℕ (isFirst (λ n → f n ≡ zero))
extract-first-root f E =
  ‖‖-rec
    (first-isProp P (λ n → ℕ-isSet (f n) 0))
    (λ { (n , Pn) → find-first P (λ n → f n ≟ 0) n Pn})
    E
  where
    P : ℕ → Type₀
    P n = f n ≡ zero
```

Finally, we can conclude with our root extraction procedure:

```
extract-root : (f : ℕ → ℕ) →
               ∃ ℕ (λ n → f n ≡ zero) →
               Σ ℕ (λ n → f n ≡ zero)
extract-root f E with extract-first-root f E
extract-root f E | n , Pn , _ = n , Pn
```

Relationship with double negation. Given a type A, the type $\neg\neg A$ is a proposition (as is the negation of any type) and there is a canonical map from the former to the later:

```
¬¬-trunc : ∀ {i} {A : Type i} → A → ¬ (¬ A)
¬¬-trunc x k = k x
```

In this sense, double negation is very similar to propositional truncation, except that the resulting type is "classical" in the sense that it satisfies the law of elimination of double negation (or, equivalently, the excluded middle). If propositional truncation $\|A\|$ can be seen as a quotient of A (we identify all proofs), and $\neg\neg A$ can be thought of as a further quotient, making the type classical. This quotient is witnessed by the existence of a canonical function $\|A\| \to \neg\neg A$, which can be constructed by

```
‖‖-¬¬ : ∀ {i} {A : Type i} → ‖ A ‖ → ¬ ¬ A
‖‖-¬¬ = ‖‖-rec ¬-isProp (λ x ¬x → ¬x x)
```

In general, there is no converse map. In particular, for a proposition A, the existence of such a map is equivalent to the type being "classical", i.e. satisfying the elimination of double negation:

```
¬¬-‖‖ : ∀ {i} {A : Type i} → isProp A →
        (¬ (¬ A) → ‖ A ‖) ↔ (¬ (¬ A) → A)
¬¬-‖‖ PA = (λ f ¬¬a → ‖‖-rec PA id (f ¬¬a)) ,
           (λ nne ¬¬a → | nne ¬¬a |)
```

Thus, if we assume that the logic is classical, in the sense that every proposition satisfies NNE, propositional truncation can be defined as double negation, see [KECA16] and [Uni13, Exercise 3.14].

Impredicative definition. Instead of defining propositional truncation axiomatically, we can almost encode it in the following way [Uni13, Exercise 3.15]:

```
‖_‖ : ∀ {i} (A : Type i) → Type i
‖_‖ {i} A = {B : Type i} → isProp B → (A → B) → B
```

The propositional truncation of a type A is a type $\|A\|$ which, by definition, is such that, for every proposition B, if we have a map $A \to B$ then we have a map $\|A\| \to B$, i.e. satisfies the elimination rule ($\|\|_{\mathrm{E}}$) stated earlier. We say "almost" here because the above definition is not accepted by Agda: if A is a type at level i then the type we have defined is not at level i but at level $i + 1$, i.e. we should actually have given it the type

```
‖_‖ : ∀ {i} (A : Type i) → Type (lsuc i)
```

There are two ways out of it. The easy one is to simply disable universe checking (with the option `--type-in-type`), but this makes the logic inconsistent, see section 8.2. The other one is to adopt a principle weaker than having type in type, called *propositional resizing*, which roughly says that a proposition in the i-th universe can be seen as a proposition in the j-th universe for any i and j (including $i > j$): after all, a proposition contains at most one element (up to homotopy), so that it is reasonable to consider that size does not matter in this case.

Anyhow, with this encoding, function extensionality allows proving that the truncation is a proposition

```
‖‖-isProp : ∀ {i} {A : Type i} → isProp ‖ A ‖
‖‖-isProp = Π'-isProp (λ B → Π-isProp (λ p → Π-isProp (λ f → p)))
```

the truncation is easy to define

```
|_| : ∀ {i} {A : Type i} → A → ‖ A ‖
| x | _ f = f x
```

the recursion principle is simple to show

```
‖‖-rec : ∀ {i j} {A : Type i} {B : Type j} →
         isProp B → (A → B) → ‖ A ‖ → B
‖‖-rec p f x = x p f
```

as well is the computation principle

```
‖‖-comp : ∀ {i j} {A : Type i} {B : Type j} →
          (p : isProp B) (f : A → B) (x : A) →
          ‖‖-rec p f | x | ≡ f x
‖‖-comp p f x = refl
```

and the uniqueness principle

```
‖‖-eta : ∀ {i} {A : Type i} (p : isProp A) (x : ‖ A ‖) →
         | ‖‖-rec p id x | ≡ x
‖‖-eta p x = funext2 (λ a f → p (f (x p id)) (x a f))
```

9.4 Univalence

As indicated before, we still lack ways to prove equalities which ought to hold in our geometric model. We now introduce the univalence axiom, due to Voevodsky, which fixes this in a satisfactory way.

9.4.1 Operations with paths. In this section, we describe some operations involving paths, which will be useful in order to formulate and study univalence.

Application. The first one, called ap, states that all functions preserve paths: given a function $f : A \to B$ and a path $p : x \equiv y$ in A, we can construct a path $f(x) \equiv f(y)$ in B, sometimes abusively written $f(p)$, by "applying" (thus the name) f to p:

```
ap : ∀ {i j} {A : Type i} {B : Type j} {x y : A} →
     (f : A → B) → x ≡ y → f x ≡ f y
ap f refl = refl
```

It can also be seen as a witness for the fact that equality is a congruence (and we have already met this function under the name cong in section 6.6). This application is compatible with concatenation of paths in the sense that $f(p \cdot q) \equiv f(p) \cdot f(q)$:

```
·-ap : ∀ {i j} {A : Type i} {B : Type j} {x y z : A} →
       (f : A → B) → (p : x ≡ y) → (q : y ≡ z) →
       ap f (p · q) ≡ ap f p · ap f q
·-ap f refl q = refl
```

Similarly, if two functions are equal and we apply them to the same argument, the results will also be equal:

```
happly : ∀ {i j} {A : Type i} {B : A → Type j}
         {f g : (x : A) → B x} →
         f ≡ g → (x : A) → f x ≡ g x
happly refl x = refl
```

Transport. Given a type A, a family of types $B : A \to$ Type can be thought of as a family of spaces $B(x)$, indexed by x in A, which varies continuously in x. As an illustration, we have figured the type A below as a segment at the bottom, and the type $B(x)$ above each point x of A as a disk. In passing, the space above, consisting of all the spaces $B(x)$, thus depicts the type $\Sigma(x : A).B$.
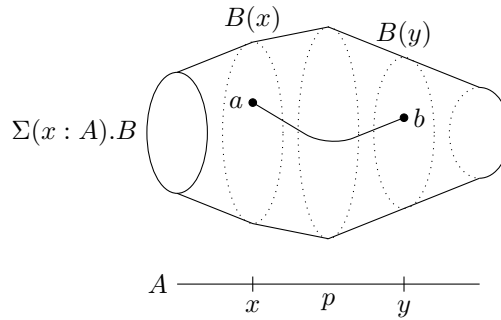
Since the spaces $B(z)$ vary continuously in z, given a path $p : x \equiv y$ in A and a point a in $B(x)$, if we make z evolve from x to y, the spaces $B(z)$ will evolve from $B(x)$ to $B(y)$ and the point a will induce a path from a to some point b in $B(y)$. We call `transport` the operation which to every path $p : x \equiv y$ and point a in $B(x)$ associates the point b in $B(y)$ resulting from "transporting" the point a in B along the path p. Formally, it can be defined as

```
transport : ∀ {i j} {A : Type i} {x y : A} (B : A → Type j) →
            x ≡ y → B x → B y
transport B refl x = x
```

This can also be seen as the fact that equality is substitutive, meaning that, in a type, we can replace an element by an equal one, and we have already encountered this function under the name `subst` in section 6.6.

This transport function allows us to define a coercion function as a particular case: if two types A and B are equal (witnessed by a path p) then we can always transform an element of type A into an element of type B by transporting an element of A into an element of B in the family where the indexing type is Type, with the type A above each point A of Type:
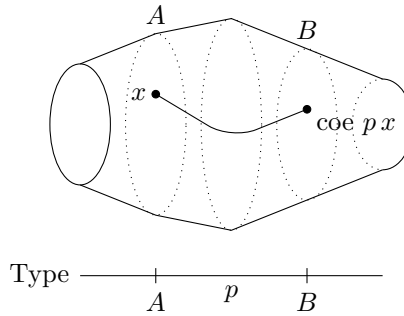
Formally,

```
coe : ∀ {i} {A B : Type i} → (A ≡ B) → A → B
coe p x = transport (λ A → A) p x
```

Of course, it could also be defined directly by induction by

```
coe : ∀ {i} {A B : Type i} → (A ≡ B) → A → B
coe refl x = x
```

Finally, we can define a variant of `ap` in the case where f is a dependent function, i.e. its type is of the form $\Pi(x : A).B(x)$. Given such a function and a path $p : x \equiv y$ in A, we cannot expect to have $f(x) \equiv f(y)$ anymore because this type does not even make sense: $f(x)$ belongs to $B(x)$ and $f(y)$ belongs to $B(y)$, so that we cannot compare them for equality. What we can show however is that if we transport $f(x)$ along p in $B(y)$ then the resulting element of $B(y)$ is

equal to $f(y)$:
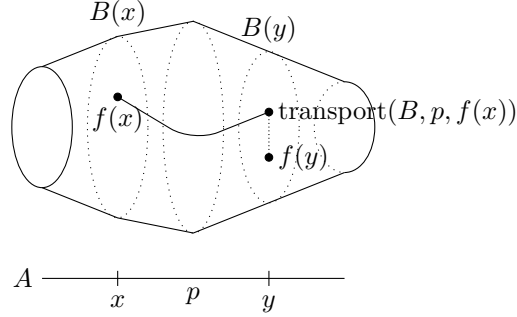
The intuitive reason for this is that f has to be a continuous function from A to B. Formally,

```
apd : ∀ {i j} {A : Type i} {B : A → Type j} {x y : A} →
      (f : (x : A) → B x) → (p : x ≡ y) →
      transport B p (f x) ≡ f y
apd f refl = refl
```

9.4.2 Equivalences. We consider that two spaces are equivalent when they are "isomorphic up to homotopy", i.e. they are homotopy equivalent, in the sense defined in section 9.2. We now formalize this notion, see [Uni13, Chapter 4] for details. We will see that it behaves much like the notion of isomorphism.

Quasi-invertibility and homotopy equivalences. Recall that two functions f and g of type $A \to B$ are *homotopic*, what we write $f \sim g$, when $f(x) \equiv g(x)$ for every point x of A. Formally, this can be defined as

```
_~_ : ∀ {i j} {A : Type i} {B : A → Type j}
      (f g : (x : A) → B x) → Type (lmax i j)
_~_ f g = ∀ x → f x ≡ g x
```

Also recall that a function $f : A \to B$ is a *homotopy equivalence* when there exists a function $g : B \to A$ such that $g \circ f \sim \mathrm{id}_A$ and $f \circ g \sim \mathrm{id}_B$. This suggests defining the predicate isQinv such that isQinv(f) holds when f is a homotopy equivalence in this sense:

```
isQinv : ∀ {i j} {A : Type i} {B : Type j} →
         (A → B) → Type (lmax i j)
isQinv {A = A} {B = B} f =
  Σ (B → A) (λ g → (g o f) ~ id × (f o g) ~ id)
```

The name comes from the fact that, a function f satisfying this property is, in this context, said to be *quasi-invertible*. Above, the identity is defined as

```
id : ∀ {i} {A : Type i} → A → A
id x = x
```

and the composition by

```
_o_ : ∀ {i j k} {A : Type i} {B : Type j} {C : Type k} →
      (B → C) → (A → B) → A → C
(g o f) x = g (f x)
```

Surprisingly, this definition turns out not to be a good one because it is not a proper predicate: isQinv(f) is not a proposition in general (see [Uni13, Theorem 4.1.3] for a counter-example) and being a quasi-inverse is thus not a property of a function f, it involves more data. We can come up with a simple variant of this definition which actually is a predicate: instead of requiring that the left and the right inverse are the same, we leave the possibility for them to be different. We say that a function $f : A \to B$ is an *equivalence* when there exists $g : B \to A$ and $g' : B \to A$ such that $g \circ f \sim \mathrm{id}_A$ and $f \circ g' \sim \mathrm{id}_B$. In Agda,

```
isEquiv : ∀ {i j} {A : Type i} {B : Type j} →
          (A → B) → Type (lmax i j)
isEquiv {A = A} {B = B} f =
  Σ (B → A) (λ g → (g o f) ~ id) ×
  Σ (B → A) (λ g → (f o g) ~ id)
```

and one can show that isEquiv(f) is a proposition for every function f [Uni13, Theorem 4.2.13]. Note that every quasi-invertible map is canonically an equivalence:

```
isQinv-isEquiv : ∀ {i j} {A : Type i} {B : Type j} {f : A → B} →
                 isQinv f → isEquiv f
isQinv-isEquiv (g , gf , fg) = (g , gf) , (g , fg)
```

There is also a converse map (which is not obvious to define), the subtle point being that the resulting pair of maps does not form an equivalence. That being said, all the equivalences we will construct in practice will be quasi-inverses.

Contractibility. The notion of equivalence can be thought of as an "up-to-homotopy" version of the notion of bijection in set theory. We can therefore try to mimic the usual characterization of bijections: a function $f : A \to B$ is a bijection when every element y in B has a unique preimage under f, i.e. $f^{-1}(y)$ is a singleton. In homotopy type theory, the analogue of the notion of preimage is given by the *fiber* of f at y which is the space of points x in A equipped with a path from $f(x)$ to y:

```
fib : ∀ {i j} {A : Type i} {B : Type j} →
      (A → B) → B → Type (lmax i j)
fib {A = A} f y = Σ A (λ x → f x ≡ y)
```

We then say that a map is *contractible* when all its fibers are:

```
isContrMap : ∀ {i j} {A : Type i} {B : Type j} →
             (A → B) → Type (lmax i j)
isContrMap {B = B} f = (y : B) → isContr (fib f y)
```

It can be shown, for a map f, that the types isEquiv(f) and isContrMap(f) are equivalent, so that we could use contractibility as an alternative definition for being an equivalence.

Equivalence of types. Two types A and B are *equivalent* when there is an equivalence from A to B, what we write $A \simeq B$:

```
_≃_ : ∀ {i j} (A : Type i) (B : Type j) → Type (lmax i j)
A ≃ B = Σ (A → B) isEquiv
```

This relation is an equivalence relation. It is reflexive:

```
≃-refl : ∀ {i} {A : Type i} → A ≃ A
≃-refl = id , (id , (λ x → refl)) , (id , λ x → refl)
```

transitive:

```
≃-trans : ∀ {i j k} {A : Type i} {B : Type j} {C : Type k} →
          A ≃ B → B ≃ C → A ≃ C
≃-trans (f , (g , gf) , (g' , fg')) (h , (i , ih) , (i' , hi')) =
  (h o f) ,
  (((g o i) , λ x → trans (ap g (ih (f x))) (gf x)) ,
  ((g' o i') , λ x → trans (ap h (fg' (i' x))) (hi' x)))
```

but also symmetric, which is not obvious because the definition of equivalence is not:

```
≃-sym : ∀ {i j} {A : Type i} {B : Type j} → A ≃ B → B ≃ A
≃-sym {B = B} (f , (g , gf) , (g' , fg')) =
  g , (f , left) , (f , gf)
  where
  g-g' : (x : B) → g x ≡ g' x
  g-g' x = trans (sym (ap g (fg' x))) (gf (g' x))
  left : (x : B) → f (g x) ≡ x
  left x = trans (ap f (g-g' x)) (fg' x)
```

An equivalence e consists of a map $f : A \to B$ together with two maps $g, g' : B \to A$ which are respectively left and right inverse for f. We can define a function which to such an equivalence associates the corresponding f:

```
≃-→ : ∀ {i j} {A : Type i} {B : Type j} → A ≃ B → A → B
≃-→ (f , _) = f
```

and one associating the corresponding g:

```
≃-← : ∀ {i j} {A : Type i} {B : Type j} → A ≃ B → B → A
≃-← (_ , ((g , _) , _)) = g
```

It will also be useful to have a notation for the proof that g is a left inverse for f, i.e. $x = g(f(x))$ for every x in A:

```
≃-η : ∀ {i j} {A : Type i} {B : Type j}
      (e : A ≃ B) (x : A) → x ≡ ≃-← e (≃-→ e x)
≃-η (f , (g , gl) , (h , hr)) x = sym (gl x)
```

We also define one providing a proof that g is a right inverse for f, i.e. we have $f(g(x)) = x$ for every x in A:

```
≃-ε : ∀ {i j} {A : Type i} {B : Type j}
      (e : A ≃ B) (y : B) → ≃-→ e (≃-← e y) ≡ y
≃-ε (f , (g , gl) , (h , hr)) y =
  f (g y)           ≡⟨ sym (ap (λ y → f (g y)) (hr y)) ⟩
  f (g (f (h y)))   ≡⟨ ap f (gl (h y)) ⟩
  f (h y)           ≡⟨ hr y ⟩
  y                 ∎
```

Note that the proof is slightly more complicated than the previous one because we show here that g and not g' is a right inverse for f.

Finally, we show a last useful theorem. In set theory, a function $f : A \to B$ which is bijective, i.e. which admits an inverse g, is always injective. This means that for every elements x and y of A, if $f(x) = f(y)$ then $x = y$. Namely, we have

$$x = g(f(x)) = g(f(y)) = y$$

This property also holds in our context:

```
≃-inj : ∀ {i j} {A : Type i} {B : Type j}
        (e : A ≃ B) {x y : A} → ≃-→ e x ≡ ≃-→ e y → x ≡ y
≃-inj e {x} {y} p =
  x                   ≡⟨ ≃-η e x ⟩
  ≃-← e (≃-→ e x)    ≡⟨ ap  (≃-← e) p ⟩
  ≃-← e (≃-→ e y)    ≡⟨ sym (≃-η e y) ⟩
  y                   ∎
```

9.4.3 Univalence. We can easily define a function which shows that two equal types A and B are equivalent:

```
id-to-equiv' : ∀ {i} {A B : Type i} → (A ≡ B) → (A ≃ B)
id-to-equiv' refl = id , ((id , (λ _ → refl)) , id , (λ _ → refl))
```

In words, by induction on the equality $p : A \equiv B$ we can suppose that A and B are the same, and in this case we can take the identity function as equivalence between the two types, left and right inverses being the identity. Given a path $p : A \equiv B$, note that the induced function $A \to B$ is precisely given by coe f, so that it is conceptually better to define this operator as

```
id-to-equiv : ∀ {i} {A B : Type i} → (A ≡ B) → (A ≃ B)
id-to-equiv p = coe p , coe-isEquiv p
```

where the proof that coercion gives rise to equivalences is

```
coe-isEquiv : ∀ {i} {A B : Type i} (p : A ≡ B) → isEquiv (coe p)
coe-isEquiv refl = (id , (λ x → refl)) , (id , λ x → refl)
```

The *univalence* axiom introduced by Voevodsky states that this function is itself an equivalence [Uni13, section 2.10]:

```
postulate univalence : ∀ {i} {A B : Type i} →
                       isEquiv (id-to-equiv {i} {A} {B})
```

i.e. we have an equivalence

```
ua-equiv : ∀ {i} {A B : Type i} → (A ≡ B) ≃ (A ≃ B)
ua-equiv = id-to-equiv , univalence
```

One of the main consequences of this axiom is that, since the types $A \equiv B$ and $A \simeq B$ are equivalent, there is a map

$$A \simeq B \to A \equiv B$$

which allows constructing a proof of equality from an equivalence:

```
ua : ∀ {i} {A B : Type i} → (A ≃ B) → (A ≡ B)
ua f = ≃-↤ ua-equiv f
```

This map can be seen as the proper introduction rule for equality, the elimination rule being `id-to-equiv`. The associated computation rule is

```
ua-comp : ∀ {i} {A B : Type i} (e : A ≃ B) → coe (ua e) ≡ (fst e)
ua-comp {A = A} {B = B} e = ap fst (≃-ε ua-equiv e)
```

and uniqueness rule is

```
ua-ext : ∀ {i} {A B : Type i} {p : A ≡ B} → p ≡ ua (id-to-equiv p)
ua-ext {p = p} = ≃-η ua-equiv p
```

Note that when A and B are types at level i, the type $A \simeq B$ is also at level i whereas $A \equiv B$ is at level $i + 1$. It is therefore crucial that we allow equivalences to hold between types at different levels, which is why we really had to properly take care of universe levels in the developments in this chapter.

9.4.4 Applications of univalence. The way univalence is quite often used is the following. It may happen that we have two different descriptions A and A' of a same data. In this case, these types can be shown to be equivalent and thus equal by ua. Since they are equal they can be used interchangeably: by transport, we can always convert a property on one into a property on the other.

For instance, the coproduct type $A \sqcup B$ can alternatively be defined as the type

$$\Sigma(b : \mathrm{Bool}).\delta_{A,B}\, b$$

where $\delta_{A,B} : \mathrm{Bool} \to \mathrm{Type}$ is the function such that $\delta_{A,B}$ false $= A$ and $\delta_{A,B}$ true $= B$. This means that we can describe an element of $A \sqcup B$ as a pair (b, x) where b is a boolean and x is an element of A (resp. B) when A is false (resp. true). An equivalence

$$(A \sqcup B) \simeq (\Sigma(b : \mathrm{Bool}).\delta_{A,B}\, b)$$

is easily constructed, from which we can deduce

$$(A \sqcup B) \equiv (\Sigma(b : \mathrm{Bool}).\delta_{A,B}\, b)$$

meaning that we can convert any property on one representation into a property on the other representation. Similarly, the type $A \times B$ can be described as the type

$$(A \times B) \equiv (\Pi(b : \mathrm{Bool}).\delta_{A,B}\, b)$$

As a more programming-oriented example, natural numbers can either be defined in unary or binary representation, giving rise to equivalent types. By univalence, we can automatically transport any operation on one representation (e.g. addition) into the other.

9.4.5 Describing identity types. Using univalence, we can describe the identity types for most type constructions.

Identity types in products. Given types A and B, we expect that a path in $A \times B$ consists of a pair of paths in A and B respectively, i.e. given x, x' in A and y, y' in B, we should have

$$\text{Id}_{A \times B}((x, y), (x', y')) \equiv \text{Id}_A(x, x') \times \text{Id}_B(y, y')$$

By univalence, this amounts to showing that the corresponding equivalence between types

$$\text{Id}_{A \times B}((x, y), (x', y')) \simeq \text{Id}_A(x, x') \times \text{Id}_B(y, y')$$

which is easily constructed:

```
×-≃ : ∀ {i j} {A : Type i} {B : Type j} {x y : A × B} →
      (x ≡ y) ≃ ((fst x ≡ fst y) × (snd x ≡ snd y))
×-≃ {x = x} {y = y} =
  f , (g , λ { refl → refl }) , (g , λ { (refl , refl) → refl })
  where
  f : x ≡ y → (fst x ≡ fst y) × (snd x ≡ snd y)
  f refl = refl , refl
  g : (fst x ≡ fst y) × (snd x ≡ snd y) → x ≡ y
  g (refl , refl) = refl
```

Identity types over natural numbers. For data types, similar characterizations can be achieved. For instance, for natural numbers, we expect that there is one proof of equality in $\text{Id}_{\mathbb{N}}(n, n)$ for any natural number n and none in $\text{Id}_{\mathbb{N}}(m, n)$ for $m \neq n$. In other words, we expect $\text{Id}_{\mathbb{N}}(n, n) = \top$ and $\text{Id}_{\mathbb{N}}(m, n) = \bot$ for $m \neq n$. We can therefore code the expected type for identity types between any two natural numbers as

```
code : ℕ → ℕ → Type₀
code zero     zero    = ⊤
code zero     (suc n) = ⊥
code (suc m)  zero    = ⊥
code (suc m)  (suc n) = code m n
```

By univalence, in order to show that natural numbers have the expected identity types, it is enough to show that there is an equivalence

$$\text{Id}_{\mathbb{N}}(m, n) \simeq \text{code } m\, n$$

To this aim we define an *encoding* function

```
enc : {m n : ℕ} → m ≡ n → code m n
enc {zero}  {.zero}    refl = tt
enc {suc n} {.(suc n)} refl = enc {n} {n} refl
```

and a *decoding* function in the other direction

```
dec : {m n : ℕ} → code m n → m ≡ n
dec {zero}  {zero}  tt = refl
dec {suc m} {suc n} c  = ap suc (dec c)
```

and finally show that they form an equivalence:

```
ℕ-eq : (m n : ℕ) → (m ≡ n) ≃ code m n
ℕ-eq m n =
  enc , ((dec , dec-enc) , (dec , enc-dec {m}))
  where
  dec-enc : {m n : ℕ} → (p : m ≡ n) → dec (enc p) ≡ p
  dec-enc {zero}  {.zero}    refl = refl
  dec-enc {suc m} {.(suc m)} refl = ap (ap suc) (dec-enc refl)
  enc-suc : {m n : ℕ} → (p : m ≡ n) → enc (ap suc p) ≡ enc p
  enc-suc refl = refl
  enc-dec : {m n : ℕ} → (c : code m n) → enc (dec {m} c) ≡ c
  enc-dec {zero} {zero} tt = refl
  enc-dec {suc m} {suc n} c =
    trans (enc-suc (dec {m} {n} c)) (enc-dec {m} {n} c)
```

9.4.6 Describing propositions. In this section, we use univalence to show that a proposition is either $\bot$ or $\top$. First, we expect that $\bot$ is the only empty type, i.e. that for every type A such that $\neg A$ holds, $A \equiv \bot$. By univalence, this amounts to showing $\neg A \to (A \simeq \bot)$, which is easily done: the map $A \to \bot$ is given by the argument $\neg A$ and the map $\bot \to A$ is given by the elimination of $\bot$. In Agda,

```
¬-≃-⊥ : ∀ {i} {A : Type i} → ¬ A → A ≃ ⊥
¬-≃-⊥ k = k ,
  (⊥-elim , λ x → ⊥-elim (k x)) ,
  (⊥-elim , λ x → ⊥-isProp _ _)
```

Similarly, $\top$ is the only contractible type, in the sense that any contractible type is equivalent to $\top$:

```
Contr-≃-⊤ : ∀ {i} {A : Type i} → isContr A → A ≃ ⊤
Contr-≃-⊤ {A = A} (x , p)  =
  f , ((g , λ y → p y) , (g , λ { tt → refl }))
  where
  f : A → ⊤
  f _ = tt
  g : ⊤ → A
  g _ = x
```

Moreover, any non-empty proposition is contractible:

```
aProp-isContr : ∀ {i} {A : Type i} → isProp A → A → isContr A
aProp-isContr PA x = x , (PA x)
```

From there, one easily deduces that a proposition is either $\bot$ when empty

```
Prop-≃-⊥ : ∀ {i} {A : Type i} → isProp A → ¬ A → A ≃ ⊥
Prop-≃-⊥ PA k = ¬-≃-⊥ k
```

or ⊤ when non-empty

```
Prop-≃-T : ∀ {i} {A : Type i} → isProp A → A → A ≃ T
Prop-≃-T PA x = Contr-≃-T (aProp-isContr PA x)
```

Decidable propositions. Above, since the logic is not classical, we need to be provided with a proof of ¬*A* or a proof of *A* in order to decide whether the proposition *A* is ⊥ or ⊤. But it is not the case that every proposition is either ⊥ or ⊤, i.e. that Prop ≡ Bool. However, this does hold for propositions which are decidable:

```
dec-Prop : ∀ {i} → Σ (Type i) (λ A → isProp A ∧ isDec A) ≃ Bool
```

(the proof is left as an exercise to the reader). By univalence, the above equivalence can be turned into an equality, thus providing a conceptually much better definition of booleans than the type with two elements: booleans is the type of decidable propositions!

Truncation of propositions. Finally, we mention that propositional truncation is idempotent on propositions, meaning that $\|A\| \equiv A$ when *A* is a proposition. By univalence, this amounts to showing $\|A\| \simeq A$: the map $\|A\| \to A$ is given by elimination of truncation and the map $A \to \|A\|$ is truncation.

```
trunc-prop : ∀ {i} {A : Type i} → isProp A → ∥ A ∥ ≃ A
trunc-prop P =
  ∥∥-rec P id ,
  (|_| , λ _ → ∥∥-isProp _ _) ,
  (|_| , λ _ → P _ _)
```

Describing contractible types. In a similar way as we have been able to describe all propositions as being either ⊥ or ⊤, we can of course characterize contractible types as being ⊤. Namely, one can show that any contractible type is equivalent to ⊤:

```
Contr-≃-T : ∀ {i} {A : Type i} → isContr A → A ≃ T
Contr-≃-T {A = A} (x , p)  =
  f , ((g , λ y → p y) , (g , λ { tt → refl }))
  where
  f : A → T
  f _ = tt
  g : T → A
  g _ = x
```

from which any contractible type can be shown to be equal to ⊤ by univalence. In other words, ⊤ is the only contractible type.

9.4.7 Incompatibility with set theoretic interpretation. The axiom of univalence forces types to act as spaces. If we assume this axiom, then it cannot be the case that we think of them as spaces but they are secretly sets: some of them have to exhibit non-trivial geometric structure. In order to show this, we shall first show that there is at least one type which is not a set: Type.

Namely, we are going to show that it contains an element, namely the type Bool of booleans, which has a non-trivial loop (i.e. a path from Bool to Bool), whereas in a set every loop has to be equal to the identity path.

A non-trivial path. Consider the negation operator not : Bool $\to$ Bool on booleans, which sends false to true and vice versa. This function is easily shown to be involutive: applying negation twice gets us back to the boolean we started with.

```
not-involutive : (b : Bool) → not (not b) ≡ b
not-involutive false = refl
not-involutive true  = refl
```

From there, we can show that negation induces an equivalence from boolean to themselves:

```
not-≃ : Bool ≃ Bool
not-≃ = not , (not , not-involutive) , (not , not-involutive)
```

Of course, we have seen that equivalence is reflexive, so that we have $A \simeq A$ for every type A, including $A =$ Bool, but the equivalence Bool $\simeq$ Bool is non-trivial, in the sense that it exchanges false and true. By univalence, this equivalence will induce a path

$$p : \mathrm{Bool} \equiv \mathrm{Bool}$$

which will not be the identity path. Geometrically, we can picture the situation as follows. The type Bool is a point in the space of all types, which contains a loop p on it induced by negation:
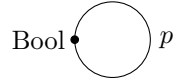
If we assume that Type is a set, then we will assimilate this path to the identity path, which will lead to a contradiction, because it will also force us to identify false and true, which we know is not the case:

```
false≢true : ¬ (false ≡ true)
false≢true ()
```

Namely, the function coe p : Bool $\to$ Bool transports a boolean along p, and the computation rule for univalence tells us that it is precisely negation. Now, if we assume that Type is a set, the path p will be equal to the path refl : Bool $\to$ Bool and therefore, we will have coe $p \equiv$ coe refl, i.e. the boolean negation function is equal to the identity. If we apply both to true (using happly), we get that false is equal to true, hence a contradiction.

```
Type-isn'tSet : ¬ (isSet Type₀)
Type-isn'tSet S = false≢true (
  false ≡⟨ happly (ap coe (S Bool Bool refl (ua not-≃))) false ⟩
  coe (ua not-≃) false ≡⟨ happly (ua-comp not-≃) false ⟩
  true ∎
  )
```

Incompatibility of UIP with univalence. As an immediate consequence the uniqueness of identity proofs principle

```
UIP : ∀ {i} → Type (lsuc i)
UIP {i} = {A : Type i} {x y : A} → (p q : x ≡ y) → p ≡ q
```

is inconsistent with univalence because it forces every type to be a set:

```
¬UIP : (∀ {i} → UIP {i}) → ⊥
¬UIP uip = Type-isn'tSet (λ x y → uip)
```

Incompatibility of double negation elimination with univalence. For similar reasons, we cannot suppose that, for every type A, we have

$$\neg\neg A \to A$$

see [Uni13, Theorem 3.2.2]. Intuitively, supposing this amounts to supposing that we have picked a particular element in every non-empty type, and this cannot reasonably be done in a continuous way.

In more details, suppose that we have a function

$$\text{nne} : \Pi(A : \text{Type}).\neg\neg A \to A$$

and write f for nne Bool : $\neg\neg$ Bool → Bool. We can easily construct an element u of $\neg\neg$ Bool (this amounts to showing that Bool is non-empty), from which we can construct an element $b = f\, u$ of Bool and we can show that not $b \equiv b$, from which we are of course able to derive a contradiction. In order to show the equality not $b \equiv b$, the main idea is, as before, to transport f along the non-trivial path $p : \text{Bool} \equiv \text{Bool}$. The resulting function when applied to u can be shown to be both equal to $f\, u$ and not $(f\, u)$.

```
¬NNE : (∀ {i} (A : Type i) → ¬ (¬ A) → A) → ⊥
¬NNE nne = not-≢ (f u) (
  not (f u)
    ≡⟨ sym (happly (ua-comp not-≃) (f u)) ⟩
  transport (λ A → A) p (f u)
    ≡⟨ ap (coe p) (ap f (¬¬-isProp u _)) ⟩
  transport (λ A → A) p (nne Bool (transport (λ A → ¬ (¬ A)) (! p) u))
    ≡⟨ sym (happly (transport→ p (λ A → ¬ (¬ A)) (λ A → A) f) u) ⟩
  transport (λ A → ¬ (¬ A) → A) p f u
    ≡⟨ happly (apd nne p) u ⟩
  f u ∎)
  where
  u : ¬ (¬ Bool)
  u k = k false
  f : ¬ (¬ Bool) → Bool
  f = nne Bool
  p : Bool ≡ Bool
  p = ua not-≃
```

Another way to prove this consists in using Hedberg's theorem presented in section 9.3.2. Namely, supposing that every type has the double negation property amounts to supposing that every type is decidable. In particular, any type should have decidable equality and thus be a set by Hedberg's theorem. But we have shown above that Type is not a set, contradiction:

```
¬NNE : (∀ {i} (A : Type i) → ¬ (¬ A) → A) → ⊥
¬NNE nne = Type-isn'tSet (Hedberg (λ x y → nne (x ≡ y)))
```

The above remark does not mean that univalence is incompatible with classical logic. It simply means that double negation elimination should be restricted to propositions if one wants to use this as an axiom, see section 9.3.1.

9.4.8 Equivalences. Univalence makes equivalences behave like equalities. We show here two instances of this which will be useful when proving function extensionality in section 9.4.9.

Firstly, when we have a function $f : A \to B$ and an equality $x \equiv y$ between elements x and y of A, we have seen that we have an induced equality $f(x) \equiv f(y)$ by the function ap of section 9.4.1. A similar property can be shown for equivalences:

```
≃-ap : ∀ {i j} {A B : Type i} (f : Type i → Type j) →
       A ≃ B → f A ≃ f B
≃-ap f e = id-to-equiv (ap f (ua e))
```

Secondly, the J rule presented in section 9.1.3 states that in order to prove a property P depending on a path p, it is enough to prove it only in the case where p is refl. This constitutes the induction principle for equalities. A similar induction principle can be shown for equivalences:

```
≃-ind : ∀ {i j} (P : {A B : Type i} → (A ≃ B) → Type j) →
        ({A : Type i} → P (≃-refl {A = A})) →
        {A B : Type i} (e : A ≃ B) → P e
≃-ind {i} P r {A} {B} e =
  transport P (≃-ε ua-equiv e) (lem (ua e))
  where
  lem : (p : A ≡ B) → P (id-to-equiv p)
  lem refl = r
```

9.4.9 Function extensionality. We have already seen in section 9.4.5 that, given types A and B, and elements x and x' in A and y and y' in B, we have

$$\mathrm{Id}_{A \times B}((x, y), (x', y')) \equiv \mathrm{Id}_A(x, y) \times \mathrm{Id}_A(x', y')$$

An equality in the product is thus the same as an equality in each of the components. Now, we have seen in section 9.4.4 that a product is a particular case of a dependent function

$$(A \times B) \equiv (\Pi(b : \mathrm{Bool}).\delta_{A,B}\, b)$$

and we therefore expect that the above characterization of paths in products generalizes to dependent functions.

More precisely, we expect that for every functions $f, g : \Pi(x : A).B$, we have

$$\mathrm{Id}_{\Pi(x:A).B}(f, g) \equiv (f \sim g)$$

i.e. the two functions f and g are equal when we have $f\,x \equiv g\,x$ for every element x of A. While we will see that this is true, the proof performed for

products above does not generalize easily. Namely, our first hope is to prove this identity using univalence, by showing an equivalence between the two types. However, this is not easy. Constructing a map from left to right is not a problem: the function `happly` defined in section 9.4.1 provides us with such a function

$$\mathrm{Id}_{\Pi(x:A).B}(f,g) \to (f \sim g)$$

However, constructing a map in the other direction

$$(f \sim g) \to \mathrm{Id}_{\Pi(x:A).B}(f,g)$$

is much more difficult. It is called *function extensionality* and corresponds to the DFE axiom we have seen in section 9.1.5. Intuitively, it can be proved as follows. The maps f and g being functions, they are thus of the form $f = \lambda x.f'$ and $g = \lambda x.g'$. Moreover, for every x, we have a path $p_x : f\,x \equiv g\,x$, and thus $f' \equiv g'$. By induction on this path, we can suppose that f' and g' are the same, in which case we can conclude that the two functions are equal by reflexivity. So, it seems that this could be proved using the following Agda code:

```
hcontr : ∀ {i j} {A : Type i} {B : A → Type j}
         (f g : (x : A) → B x) → f ~ g → f ≡ g
hcontr (λ x → f') (λ x → g') h with h x
hcontr (λ x → f') (λ x → .f') | refl = refl
```

Unfortunately, this code is not accepted by Agda: it does not allow pattern matching on functions (for good reasons) and we use an "arbitrary variable" `x` when matching on `h x`, which is not valid either.

General approach. Instead, the trick is to show the equality for all pairs of functions f and g at once, i.e. show

$$\Sigma(f : \Pi(x:A).B).\Sigma(g : \Pi(x:A).B).\mathrm{Id}_{\Pi(x:A).B}(f,g)$$
$$\equiv$$
$$\Sigma(f : \Pi(x:A).B).\Sigma(g : \Pi(x:A).B).f \sim g$$

We are thus lead to consider the type

$$\mathrm{Path}(A) = \Sigma(x:A).\Sigma(y:A).\mathrm{Id}_A(x,y)$$

of all paths in a type A and the type

$$\mathrm{Homotopy}(A,B) = \Sigma(f : \Pi(x:A).B).\Sigma(g : \Pi(x:A).B).f \sim g$$

of all homotopies between functions from a type A to a type B. A homotopy between a function f and a function g, is a function which to every x in A associates a path between $f(x)$ and $g(x)$, so that the type of all homotopies between functions from A to B can alternatively be described as

$$\mathrm{Homotopy}(A,B) = \Pi(x:A).\mathrm{Path}(B(x))$$

We will adopt this definition since it leads to simpler developments.

For every type A, we have a function $\mathrm{Path}(A) \to A$ which associates its source to a path and a function $A \to \mathrm{Path}(A)$ which constructs the constant

path on an element of A. These two can be shown to form an equivalence, i.e. we have

$$\text{Path}(A) \simeq A$$

We therefore have the following sequence of equivalences

$$\text{Homotopy}(A, B) = \Pi(x : A). \text{Path}(B) \simeq \Pi(x : A).B \simeq \text{Path}(\Pi(x : A).B)$$

and in particular, we have a map

$$\text{funext} : \text{Homotopy}(A, B) \to \text{Path}(\Pi(x : A).B)$$

which witnesses the extensionality of functions. At least, this is the general plan: if we look at this proof in details, there are some problems with it.

Firstly, the map funext above associates to each homotopy h between functions f and g a path funext(h), but we have not shown yet that this path actually also is between f and g and not some other functions. Here is how we are going to prove it. In the type Homotopy(A, B), apart from h, there is another notable homotopy, which we write h_0 here: the "constant" homotopy between f and itself. It can be shown that funext(h_0) = funext(h), the proof being just reflexivity, and therefore $h_0 = h$ because funext is an equivalence, and as such is injective. We have an equality between a homotopy $f \sim f$ and a homotopy $f \sim g$: by projecting on the target endpoint, we can deduce that $f = g$.

Secondly, the middle equivalence

$$\Pi(x : A). \text{Path}(B) \simeq \Pi(x : A).B$$

is easy to prove only when B does not depend on x. Namely, we have an equivalence Path$(B) \simeq B$: if B does not depend on x, we can simply apply the function $\lambda B.(A \to B)$ to it in order to obtain the desired equivalence. If B depends on x, there is no easy way to proceed, at least if we do not suppose function extensionality, which is precisely what we are trying to prove. The plan will thus be to proceed in three steps:

1. show function extensionality in the non-dependent case as above,

2. use it to deduce another property called weak function extensionality,

3. use weak function extensionality to deduce dependent function extensionality.

Paths. Let us first define the type Path(A) of all paths in a type A, as well as simple helper functions. This type can be formalized in Agda as

```
Path : ∀ {i} → (A : Type i) → Type i
Path A = Σ A (λ x → Σ A (λ y → x ≡ y))
```

We can define a function which to a path associates its source:

```
Path-src : ∀ {i} {A : Type i} → Path A → A
Path-src (x , y , p) = x
```

and its target:

```
Path-tgt : ∀ {i} {A : Type i} → Path A → A
Path-tgt (x , y , p) = y
```

so that each path induces an equality from its source to its target:

```
Path-≡ : ∀ {i} {A : Type i} → (p : Path A) →
         Path-src p ≡ Path-tgt p
Path-≡ (x , y , p) = p
```

Every identity can be seen as a path:

```
Path-of : ∀ {i} {A : Type i} {x y : A} → (p : x ≡ y) → Path A
Path-of {x = x} {y = y} p = x , y , p
```

and we can easily construct constant paths:

```
Path-cst : ∀ {i} {A : Type i} → A → Path A
Path-cst x = Path-of (refl {x = x})
```

Since every element of Path A consists of a path $x \equiv y$, we can "contract" each of its elements to its source x and show the equivalence that we have already mentioned:

```
Path-contract : ∀ {i} {A : Type i} → Path A ≃ A
Path-contract =
  (Path-src ,
    (Path-cst , λ { (_ , _ , refl) → refl }) ,
    (Path-cst , λ _ → refl))
```

Homotopies. We now define the type Homotopy(A, B) of all homotopies between dependent functions from A to B:

```
Homotopy : ∀ {i j} (A : Type i) (B : A → Type j) → Type (lmax i j)
Homotopy A B = (x : A) → Path (B x)
```

The source function of the homotopy can be recovered by

```
Homotopy-src : ∀ {i j} {A : Type i} {B : A → Type j} →
               Homotopy A B → (x : A) → B x
Homotopy-src h x = Path-src (h x)
```

and similarly for its target

```
Homotopy-tgt : ∀ {i j} {A : Type i} {B : A → Type j} →
               Homotopy A B → (x : A) → B x
Homotopy-tgt h x = Path-tgt (h x)
```

so that every homotopy induces a homotopy in the previous sense between its source and its target:

```
Homotopy-~ : ∀ {i j} {A : Type i} {B : A → Type j}
             (h : Homotopy A B) → Homotopy-src h ~ Homotopy-tgt h
Homotopy-~ h x = Path-≡ (h x)
```

We can see a homotopy between two given functions as an element of this type

```
Homotopy-of : ∀ {i j} {A : Type i} {B : A → Type j}
              {f g : (x : A) → B x} → f ~ g → Homotopy A B
Homotopy-of h x = Path-of (h x)
```

and given a function $f : A \to B$, we can construct the constant homotopy $f \sim f$:

```
Homotopy-cst : ∀ {i j} {A : Type i} {B : A → Type j} →
               ((x : A) → B x) → Homotopy A B
Homotopy-cst f = Homotopy-of (λ x → refl {x = f x})
```

Non-dependent function extensionality. Finally, we can show the promised equivalence. As explained above, we restrict here to the non-dependent case, where B does not depend on x:

```
Homotopy-≃-Path : ∀ {i j} {A : Type i} {B : Type j} →
                  Homotopy A (λ _ → B) ≃ Path (A → B)
Homotopy-≃-Path {i} {j} {A} {B} =
  (Homotopy A (λ _ → B)) ≃⟨ ≃-refl ⟩
  (A → Path B)           ≃⟨ ≃-to  Path-contract ⟩
  (A → B)                ≃⟨ ≃-sym Path-contract ⟩
  Path (A → B)           ≃∎
```

where the function ≃-to is detailed below. From there, the non-dependent function extensionality is easily deduced, its type being

```
FE {i} {j} = {A : Type i} {B : Type j} → {f g : A → B} →
             ((x : A) → f x ≡ g x) → f ≡ g
```

We can proceed as explained before, by considering the constant homotopy h_0 on f and the homotopy h between f and g, showing that they have the same image under the function ≃-→ Homotopy-≃-Path (the proof is simply refl because we have carefully defined ≃-to, see below), deducing by injectivity that $h_0 = h$ and deducing that $f = g$ by projecting on the respective targets of h_0 and h.

```
funext-nd : ∀ {i j} → FE {i} {j}
funext-nd {A = A} {B = B} {f = f} {g = g} h =
  ap (λ h x → Homotopy-tgt h x) p
  where
  p : Homotopy-cst f ≡ Homotopy-of h
  p = ≃-inj Homotopy-≃-Path refl
```

Functions to equivalent types. The core of the series of equivalences proving Homotopy-≃-Path is the function ≃-to which allows deducing

$$(A \to B) \simeq (A \to B')$$

from

$$B \simeq B'$$

This is actually the only place where the univalence axiom is used. Since we have application of functions to equivalences, this is actually pretty easy to define:

```
≃-to : ∀ {i j} → {A : Type i} → {B B' : Type j} →
       B ≃ B' → (A → B) ≃ (A → B')
≃-to {A = A} e = ≃-ap (λ B → A → B) e
```

Given a function $f : B \to B'$ which is an equivalence, we have "no control" over the function $(A \to B) \to (A \to B')$ which is the produced equivalence, which complicates the proofs. However, there would be a natural candidate, namely the function

$$\lambda g x. f(g\, x) : (A \to B) \to (A \to B')$$

It simplifies much the proofs if we enforce this choice. This can be done by defining instead:

```
≃-to : ∀ {i j} → {A : Type i} → {B B' : Type j} →
       B ≃ B' → (A → B) ≃ (A → B')
≃-to {i} {j} {A} {B} {B'} e = (λ f x → (≃-→ e) (f x)) , lem e
  where
  lem : {B B' : Type j} (e : B ≃ B') →
        isEquiv (λ (f : A → B) x → (≃-→ e) (f x))
  lem = ≃-ind
          (λ {B} e → isEquiv (λ (f : A → B) x → (≃-→ e) (f x)))
          (λ {B} → snd (≃-refl {A = A → B}))
```

Weak function extensionality. In order to generalize function extensionality to dependent types, we will first show another principle called *weak function extensionality*, which states that a product of contractible types is itself contractible. It can also be seen as a degenerated form of axiom of choice where the family of types we consider consists of types containing exactly one element (up to homotopy). Formally, it can be stated as follows:

```
WFE {i} {j} = {A : Type i} {B : A → Type j} →
              ((x : A) → isContr (B x)) → isContr ((x : A) → B x)
```

Let us first explain why the "obvious proof" does not work. Suppose given a family of contractible types $B(x)$ indexed by x in A: for each x, there is an element b_x in $B(x)$ and a path $p_x^y : b_x \equiv y$ for every y in B. We are therefore tempted to prove that $\Pi(x : A).B$ can be contracted on to $\lambda x.b_x$. To show that this is the case, we have to construct, for every function f in $\Pi(x : A).B$ a path $\lambda x.b_x \equiv f$. Since, we have the paths $p_x^{f(x)} : b_x \equiv f(x)$ we are almost there, but we cannot conclude since this would require function extensionality, which is precisely what we are trying to prove...

The actual proof uses (non-dependent) function extensionality. Suppose given a family of contractible types $B(x)$ indexed by x in A. Each $B(x)$ being contractible, we have $B(x) \simeq \top$, and thus $B(x) \equiv \top$ by univalence. Therefore, $B \equiv \lambda x.\top$ by function extensionality. By transport, instead of showing that the type $\Pi(x : A).B$ is contractible we are left with showing that the type $\Pi(x : A).\top$ is contractible, which is easy: it can be contracted to $\lambda x.\,\mathrm{tt}$.

```
wfunext : ∀ {i j} → WFE {i} {j}
wfunext {A = A} {B = B} c =
  transport (λ B → isContr ((x : A) → B x)) (sym p) contr
  where
```

```
p : B ≡ (λ _ → Lift T)
p = funext-nd (λ x → ua (Contr-≃-Lift-T (c x)))
contr : ∀ {i} → isContr ((x : A) → Lift {i} T)
contr = (λ x → lift tt) , (λ f → funext-nd (λ x → refl))
```

Function extensionality. We can finally prove the (dependent) function extensionality, whose type is

```
DFE {i} {j} =
  {A : Type i} {B : A → Type j} → {f g : (x : A) → B x} →
  ((x : A) → f x ≡ g x) → f ≡ g
```

Suppose given two dependent functions f and g of type $\Pi(x : A).B$, which are homotopic (i.e. we have $f \sim g$). Up to some minor details, those functions can be seen as elements of type

$$\Pi(x : A).\Sigma(y : B).\mathrm{Id}_B(f(x), y)$$

which we respectively call f' and g', the definition of the latter using the fact that we have a homotopy. Recall that the type $\Sigma(y : B).\mathrm{Id}_B(f(x), y)$, is what we called the singleton at $f(x)$ and is contractible, see section 9.3.3; therefore, by weak function extensionality, the above type is also contractible. The functions f' and g' being elements of a contractible type, they are necessarily equal, from which one easily deduces that f and g are equal.

```
funext : ∀ {i j} → DFE {i} {j}
funext {A = A} {B = B} {f = f} {g = g} p =
  ap (λ f x → fst (f x)) p'
  where
  f' : (x : A) → Singleton (f x)
  f' x = f x , refl
  g' : (x : A) → Singleton (f x)
  g' x = g x , p x
  contr : isContr ((x : A) → Singleton (f x))
  contr = wfunext (λ x → Singleton-isContr (f x))
  p' : f' ≡ g'
  p' = Contr-isProp contr f' g'
```

The above proof does not use univalence and therefore, without univalence, WFE implies DFE. The converse also holds, as explained above,

```
DFE-to-WFE : ∀ {i j} → DFE {i} {j} → WFE {i} {j}
DFE-to-WFE funext c =
  (λ x → fst (c x)) , λ f → funext (λ x → snd (c x) (f x))
```

so that WFE and DFE are equivalent, even without assuming univalence.

9.4.10 Propositional extensionality. Recall from section 9.3.1 that the propositional extensionality axiom states that two logically equivalent propositions A and B are equal:

```
PE : ∀ {i} → Type (lsuc i)
PE {i} = ∀ {A B : Type i} → isProp A → isProp B → A ↔ B → A ≡ B
```

This is intuitively justified because, since A and B are both propositions they are either empty or a point, and since they are equivalent they are both empty or both non-empty. We show here that this principle follows from univalence. Namely, two logically equivalent propositions A and B are equivalent: the logical equivalence provides functions $f : A \to B$ and $g : B \to A$ and we have $g \circ f(x) \equiv x$ and $f \circ g(y) \equiv y$ for every x in A and y in B because A and B are propositions (and thus any two elements are equal).

```
↔-to-≃ : ∀ {i} {A B : Type i} →
        isProp A → isProp B → A ↔ B → A ≃ B
↔-to-≃ PA PB (f , g) =
  f ,
  (g , (λ x → PA (g (f x)) x)) ,
  (g , (λ x → PB (f (g x)) x))
```

Finally, univalence provides us with the required equality:

```
propext : ∀ {i} → PE {i}
propext PA PB e = ua (↔-to-≃ PA PB e)
```

By transport, this means that given two equivalent propositions, one can be substituted for the other. We have already encountered an instance of this in lemma 2.2.9.1.

9.5 Higher inductive types

We have seen in section 9.4.7 that, if we assume the axiom of univalence, we can exhibit a type which is non-trivial, in the sense that it is not a set. However, we cannot easily construct a type which corresponds to a space we have in mind. In particular, we have mentioned in section 9.3.2 that all the usual (inductive) types are sets (e.g. natural numbers, lists of elements of a set, etc.). Higher inductive types are a generalization of inductive types that allow for constructing useful types, which are typically not sets. The presentation given here is very brief and the reader is invited to read [Uni13, chapter 6] for a more detailed presentation, as well as [CCHM16] for a technical description of the theory behind the current implementation in Agda.

9.5.1 Rules for higher types. In order to introduce types corresponding to spaces of interest, one way to proceed consists in adding new constructors and rules, as in section 8.3. We present this approach here.

The interval type. As a first example consider the *interval* space

$$\text{beg} \bullet \!\!\xrightarrow{\text{path}}\!\! \bullet \text{end}$$

This type is of course a set (and even a contractible type), but the approach will generalize to types which are not. The corresponding type, that we are going to write I, can be thought of as freely generated by two points beg and end, as well as a path path : beg $\equiv$ end, as figured above, which suggests the following

rules. The formation rule states that I is a well-formed type in any well-formed context

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathrm{I} : \mathrm{Type}} \; (\mathrm{I_F})$$

The introduction rules states that beg and end are elements of the interval and that path is a path between them:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathrm{beg} : \mathrm{I}} \; (\mathrm{I_I^{beg}}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathrm{end} : \mathrm{I}} \; (\mathrm{I_I^{end}}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathrm{path} : \mathrm{Id_I}(\mathrm{beg}, \mathrm{end})} \; (\mathrm{I_I^{path}})$$

The elimination rule is more subtle. What do we need in order to determine a function from I to an arbitrary type A? In the case where A does not depend on I, this is easy: we need two elements b and e of A (the respective images of beg and end), as well as a path p from b to e (the image of path). The corresponding rule should thus be

$$\frac{\Gamma \vdash t : \mathrm{I} \qquad \Gamma \vdash b : A \qquad \Gamma \vdash e : A \qquad \Gamma \vdash p : \mathrm{Id}_A(b, e)}{\Gamma \vdash \mathrm{rec}(t, x \mapsto A, b, e, p) : A} \; (\mathrm{I_E})$$

where $\mathrm{rec}(t, x \mapsto A, b, e, p)$ can be thought of as the image of an arbitrary point t of I when the path path is sent to p. As usual, we want to formulate this elimination rule in the more general case where A depends on I, i.e. has a free variable x of type I. We now expect b to be of type $A[\mathrm{beg}/x]$ and e of type $A[\mathrm{end}/x]$, and now we are facing a problem: we cannot state anymore that the path p should go from b to e, because b and e do not live in the same type anymore! A way to overcome this problem, and be able to compare the two points, consists in transporting the point b along path, see section 9.4.1, in order to obtain a point b' in $A[\mathrm{end}/x]$ and then require the path p to lie between b' and e.
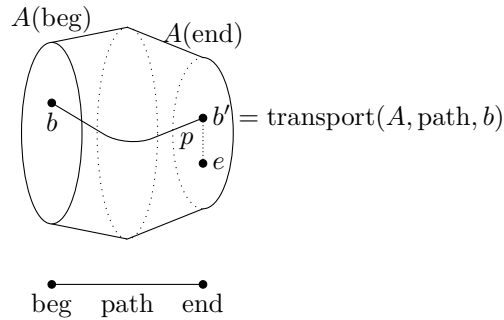
The resulting dependent elimination rule is then

$$\frac{\Gamma \vdash t : \mathrm{I} \qquad \Gamma, x : \mathrm{I} \vdash A : \mathrm{Type} \\ \Gamma \vdash b : A[\mathrm{beg}/x] \qquad \Gamma \vdash e : A[\mathrm{end}/x] \qquad \Gamma \vdash p : \mathrm{Id}_{A[\mathrm{end}/x]}(b', e)}{\Gamma \vdash \mathrm{rec}(t, x \mapsto A, b, e, p) : A[t/x]} \; (\mathrm{I_E})$$

where b' is a shorthand for $\mathrm{transport}(A, \mathrm{path}, b)$. The computation rules state that when we apply the elimination rule in the case where t is beg, end and

path, we recover b, e and p respectively:

$$\frac{\begin{array}{ccc} & \Gamma, x : \mathrm{I} \vdash A : \mathrm{Type} & \\ \Gamma \vdash b : A[\mathrm{beg}\,/x] & \Gamma \vdash e : A[\mathrm{end}\,/x] & \Gamma \vdash p : \mathrm{Id}_{A[\mathrm{end}\,/x]}(b', e) \end{array}}{\Gamma \vdash \mathrm{rec}(\mathrm{beg}, x \mapsto A, b, e, p) = b : A[\mathrm{beg}\,/x]} \;(\mathrm{I}_{\mathrm{C}}^{\mathrm{beg}})$$

$$\frac{\begin{array}{ccc} & \Gamma, x : \mathrm{I} \vdash A : \mathrm{Type} & \\ \Gamma \vdash b : A[\mathrm{beg}\,/x] & \Gamma \vdash e : A[\mathrm{end}\,/x] & \Gamma \vdash p : \mathrm{Id}_{A[\mathrm{end}\,/x]}(b', e) \end{array}}{\Gamma \vdash \mathrm{rec}(\mathrm{end}, x \mapsto A, b, e, p) = e : A[\mathrm{end}\,/x]} \;(\mathrm{I}_{\mathrm{C}}^{\mathrm{end}})$$

$$\frac{\begin{array}{ccc} & \Gamma, x : \mathrm{I} \vdash A : \mathrm{Type} & \\ \Gamma \vdash b : A[\mathrm{beg}\,/x] & \Gamma \vdash e : A[\mathrm{end}\,/x] & \Gamma \vdash p : \mathrm{Id}_{A[\mathrm{end}\,/x]}(b', e) \end{array}}{\Gamma \vdash \mathrm{apd}(\mathrm{rec}(-, x \mapsto A, b, e, p), \mathrm{path}) = p : \mathrm{Id}_{A[\mathrm{end}\,/x]}(b', e)} \;(\mathrm{I}_{\mathrm{C}}^{\mathrm{path}})$$

We do not include a uniqueness rule because it can be shown to hold propositionally (this is detailed in section 9.5.3 in the case of the circle type).

The circle type. A type Circle corresponding to the *circle* can easily be implemented, if we think of the circle as being freely generated by a point, that we call base, and a path loop : base $\equiv$ base:
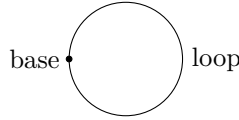
In other words, it is the above interval type, where the beginning and end point have been identified.

The formation rule states that Circle is a well-formed type in a well-formed context:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathrm{Circle} : \mathrm{Type}} \;(\mathrm{Circle}_{\mathrm{F}})$$

The introduction rules allow typing the point base and the path loop:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathrm{base} : \mathrm{Circle}} \;(\mathrm{Circle}_{\mathrm{I}}^{\mathrm{base}}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathrm{loop} : \mathrm{Id}_{\mathrm{Circle}}(\mathrm{base}, \mathrm{base})} \;(\mathrm{Circle}_{\mathrm{I}}^{\mathrm{loop}})$$

The elimination rule states that an application from the circle Circle into an arbitrary type A is determined by a point b of A (the image of base) and a path p (which determines the image of loop, as explained above):

$$\frac{\begin{array}{ccc} & \Gamma \vdash t : \mathrm{Circle} & \\ \Gamma, x : \mathrm{I} \vdash A : \mathrm{Type} & \Gamma \vdash b : A[\mathrm{base}\,/x] & \Gamma \vdash p : \mathrm{Id}_{A[\mathrm{base}\,/x]}(b', e) \end{array}}{\Gamma \vdash \mathrm{rec}(t, x \mapsto A, b, e, p) : A[t/x]} \;(\mathrm{Circle}_{\mathrm{E}})$$

where b' is a shorthand for $\mathrm{transport}(A, \mathrm{loop}, b)$. The computation rules are left to the reader. The reader should get convinced that we could write the rules for the type corresponding to the usual low dimensional spaces in this way.

Exercise 9.5.1.1. This is not the only way of implementing the circle. For instance, formalize the type corresponding to the following description of the sphere:

$$x \begin{array}{c} p \\ \bigcirc \\ q \end{array} y$$
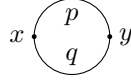
i.e. freely generated by two points x and y and two paths p and q.

Exercise 9.5.1.2. Write down the rules for the type corresponding to the sphere.

9.5.2 Paths over. As noted above, when writing the elimination rule of types involving paths as constructors one needs to compare elements (say, b and e) of distinct types (say, $A[\text{beg}/x]$ and $A[\text{end}/x]$), and the way we used to proceed consisted in transporting the first along p into b', so that it lies in the same type as the second. Here, a path between b' and e can be thought of as representing a path between b and e, i.e. as a way of comparing two elements which do not live in the same type. This is similar to what we have done in section 6.6.9 when defining heterogeneous equality, although we have to be more precise about equalities here.

Given a path $p : x \equiv y$ in a type A, a dependent type $B : A \to \text{Type}$, and two elements $t : B(x)$ and $u : B(y)$, we write $t \equiv_p^B u$ for the type of *paths over p* between t and u. This intuitively corresponds to the collection of paths between t and u whose projection onto A gives the path p:
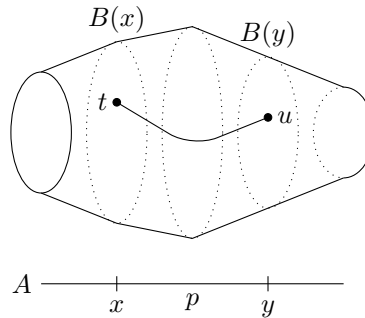
As indicated above, this type can be defined using transport

```
PathOver : ∀ {i j} {A : Type i} (B : A → Type j) {x y : A}
           (p : x ≡ y) (t : B x) (u : B y) → Type j
PathOver B p t u = (transport B p t) ≡ u
```

although it is maybe clearer (and closer to the definition of heterogeneous equality, see section 6.6.9) to define it by induction on the path p:

```
PathOver : ∀ {i j} {A : Type i} (B : A → Type j) {x y : A}
           (p : x ≡ y) (t : B x) (u : B y) → Type j
PathOver B refl t u = (t ≡ u)
```

It is convenient to introduce the following notation

```
syntax PathOver B p t u = t ≡ u [ B ↓ p ]
```

which allows writing in Agda

$$t \equiv u \; [\; B \downarrow p \;]$$

what we have been writing $t \equiv_p^B u$ earlier. This new definition could be used to simplify the types of functions in various places. For instance, the function apd, see section 9.4.1, could be defined as

```
apd : ∀ {i j} {A : Type i} {B : A → Type j} (f : (a : A) → B a)
      {x y : A} → (p : x ≡ y) → f x ≡ f y [ B ↓ p ]
apd f refl = refl
```

9.5.3 The circle as a higher inductive type. As usual in Agda, instead of implementing types of interest one by one, we expect that they are particular cases of inductive types. For instance, the circle being generated by a point and a path, we expect that it can be described by the following inductive type:

```
data Circle : Type₀ where
  base : Circle
  loop : base ≡ base
```

If you try this at home, of course Agda will reject it: as we have seen in section 8.4, all the constructors defining an inductive type A should have A as target, but here the type of the constructor loop is base ≡ base, i.e. an equality between elements of Circle i, not an element of Circle i (unlike base for instance). *Higher inductive types* are a generalization of inductive types allowing constructors of equalities between elements of the type. Defining those properly is out of scope here, we will only try to give some examples of those. An extension of Agda, called *cubical Agda*, allows for trying them by beginning our files with

```
{-# OPTIONS --cubical #-}
```

and importing the dedicated library

```
open import Cubical.Core.Everything
```

This allows in particular for the above definition to be accepted by Agda. From there, we can show the recursion principle associated to the circle type:

```
Circle-rec : ∀ {i} {A : Type i} (b : A) (p : b ≡ b) → Circle → A
Circle-rec b p base = b
Circle-rec b p (loop ι) = p ι
```

It corresponds to the elimination rule and, as explained before, formalizes the fact that a map from the circle to an arbitrary type A is determined by a point b of A (the image of base) and a path $p : b \equiv b$ (the image of loop). As it can be observed above, when we perform pattern matching on an element of the circle, Agda generates two cases: this element is either the point base or a point loop ι in the loop path. Here, the variable ι can be thought of as indexing the position where we are in the path loop: you can think of ι as being a real number between 0 and 1 such that loop 0 (resp. loop 1) is the start (resp. end) of the loop, although we will not need to understand precisely what this variable precisely means here. The induction principle, which is the dependent variant of the above can also be proved in the same way:

```
Circle-ind : ∀ {i} {A : Circle → Type i} (b : A base)
             (p : b ≡ b [ A ↓ loop ]) (x : Circle) → A x
Circle-ind b p base = b
Circle-ind b p (loop ι) = p ι
```

We have indicated that the uniqueness rule could be derived propositionally: if two maps f and g from the circle to some type A have the same (i.e. propositionally equal) image of the base and the same image of the loops then they are equal:

```
Circle-unique : ∀ {i} {A : Type i} → (f g : Circle → A) →
                (p : f base ≡ g base) →
                ap f loop ≡ ap g loop [ (λ x → x ≡ x) ↓ p ] →
                (x : Circle) → f x ≡ g x
Circle-unique f g p q base = p
Circle-unique f g p q (loop ι) ι' = q ι' ι
```

Exercise 9.5.3.1. Show that a map from the circle to a type A is the same as a loop in A, i.e. a path $p : x \equiv x$ for some point x of A:

```
Circle-path :
   ∀ {i} {A : Type i} → (Circle i → A) ≃ Σ A (λ x → x ≡ x)
```

Exercise 9.5.3.2. Define the circle as a type `Circle'` freely generated by two points and two paths between them, as explained in exercise 9.5.1.1. Show that the types `Circle` and `Circle'` are equivalent and thus equal by univalence.

The loop space of the circle. As an illustration of the use of this type and its elimination principle, let us show a fundamental theorem of homotopy theory, the fact that the type `base ≡ base` consisting of equalities from `base` to itself, or *loops*, is equivalent to $\mathbb{Z}$ (and thus equal by univalence). Namely, those paths are characterized by the number of times they turn around the circle, the sign encoding the direction of the loops. The proof follows the technique already encountered in section 9.4.5 and is detailed in [Uni13, Section 8.1]: we are going to show that we can encode the paths as elements of $\mathbb{Z}$, as well as provide an inverse decoding function. For reasons of "continuity", we cannot reason only on loops, and actually have to reason on all paths of the form `base ≡ x` for an arbitrary element x of the circle.

We first define a function `code`, which to every point x of the circle associates a type in which we can encode paths `base ≡ x`:

```
code : Circle → Set
code = Circle-rec Type₀ ℤ (ua suc-≃)
```

(we recall that the type $\mathbb{Z}$ of integers was defined in section 6.4.9). The base point is sent to $\mathbb{Z}$ for the reason explained above, and the circle is sent to the path $\mathbb{Z} \equiv \mathbb{Z}$ induced by the successor function on $\mathbb{Z}$, which is an equivalence (with predecessor as inverse function). Namely, following the loop of the circle adds one to the number of loops of a path, and indeed, we have that transporting an integer along the loop corresponds to taking its successor:

```
transport-loop : (n : ℤ) → transport code loop n ≡ suc n
```

Geometrically, the picture to have in mind is an helix standing above a circle:
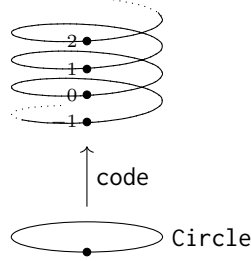
The function `code` sends each point of the circle to the set of points above it, which is isomorphic to $\mathbb{Z}$, and transporting an integer along the loop sends it to its successor.

We can encode the paths from the base point as elements of this type by transporting 0 along the path:

```
enc : (x : Circle) → base ≡ x → code x
enc x p = transport code p zero
```

Conversely, we can decode an integer as a path by the function

```
dec : (x : Circle) → code x → base ≡ x
dec = Circle-ind
  (λ x → code x → base ≡ x)
  loops
  transport-loop-loops
```

which is defined by induction on the circle. For the base case, we send an integer n to the loop of the circle concatenated n times with itself (and taking the inverse when n is negative): this path is defined by induction on n by the function

```
loops : ℤ → base ≡ base
loops (pos ℕ.zero)       = refl
loops (pos (ℕ.suc n))    = loops (pos n) · loop
loops (negsuc ℕ.zero)    = ! loop
loops (negsuc (ℕ.suc n)) = loops (negsuc n) · ! loop
```

For the loop case, we have to show that this function is invariant under transport around loop:

```
transport-loop-loops :
  transport (λ x → code x → base ≡ x) loop loops ≡ loops
```

Finally, we can show that the two functions are mutually inverse. This is purely formal on one direction:

```
dec-enc : (x : Circle) (p : base ≡ x) → dec x (enc x p) ≡ p
dec-enc .base refl = refl
```

On the other direction, this can be shown by induction on the circle:

```
enc-dec : (x : Circle) (n : code x) → enc x (dec x n) ≡ n
enc-dec = Circle-ind
  (λ x → (n : code x) → enc x (dec x n) ≡ n)
  (λ n →
    enc base (dec base n)         ≡⟨ refl ⟩
    transport code (loops n) zero ≡⟨ transport-loops n zero ⟩
    n + zero                      ≡⟨ +-unit-r n ⟩
    n                             ∎)
  (funext (λ n → ℤ-isSet _ _ _ _))
```

where

```
transport-loops : (m n : ℤ) → transport code (loops m) n ≡ m + n
```

is a generalization of `transport-loop` obtained by induction, `+-unit-r` is a proof that addition admits 0 as neutral element on the right, and `ℤ-isSet` is a proof that $\mathbb{Z}$ is a set (which follows from the decidability of equality by Hedberg theorem, see section 9.3.2).

9.5.4 Useful higher inductive types. In order to further illustrate the use of higher inductive types, we briefly present two quite useful ones: suspension and propositional truncation.

Suspension. The *suspension* ΣA of a space A is the space obtained from A by adding two new points N and S (for "north" and "south", these two points being thought of as respectively lying above and below the original space A), as well as a path going from N to S passing by x for each point x of A. For instance, starting from the space consisting of a point and segment figured on the left, we obtain the space on the right:

In particular, if iteratively apply this suspension operation starting from the empty space, we obtain the spheres:

$$\Sigma^0 \emptyset \qquad \Sigma^1 \emptyset \qquad \Sigma^2 \emptyset \qquad \Sigma^3 \emptyset \qquad \dots$$
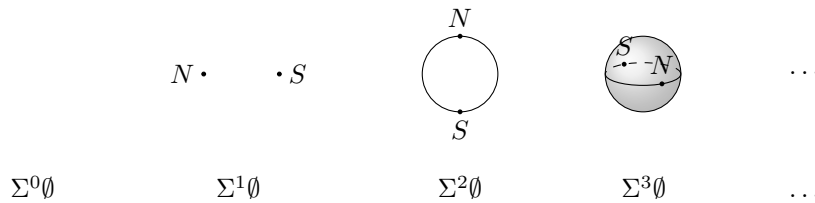
More precisely, the n-sphere is the $(n+1)$-th suspension of the empty space (the empty space could thus be considered as a good notion of (-1)-sphere). In Agda, we can define the suspension of a type as the higher inductive type

```
data Susp {i} (A : Type i) : Type i where
  N : Susp A
  S : Susp A
  p : (x : A) → N ≡ S
```

and the function which to a natural number n associates the n-sphere by

```
Sphere : ℕ → Type₀
Sphere zero    = Susp ⊥
Sphere (suc n) = Susp (Sphere n)
```

Propositional truncation. The propositional truncation operation introduced in section 9.3.4 can also be defined as a higher inductive type. In order to do so, we should recall that the propositional truncation $\|A\|$ of a type A is the type obtained from A by turning it into a proposition, i.e. by formally adding a path between any pair of points. This suggests the following definition as a higher inductive type:

```
data ‖_‖ {i} (A : Type i) : Type i where
  |_|       : A → ‖ A ‖
  ‖‖-isProp : (x y : ‖ A ‖) → x ≡ y
```

The first constructor (`|_|`) states that any point of A is a point of $\|A\|$, and the second one (`‖‖-isProp`) adds all the required paths. The resulting type is trivially a proposition by `‖‖-isProp` and the associated recursion principle, which corresponds to the elimination rule ($\|\|_E$), can be shown as follows:

```
‖‖-rec : ∀ {i j} {A : Type i} {B : Type j} →
           isProp B → (A → B) → ‖ A ‖ → B
‖‖-rec PB f | x | = f x
‖‖-rec PB f (‖‖-isProp x y ι) =
  PB (‖‖-rec PB f x) (‖‖-rec PB f y) ι
```

It can, for instance, be used to construct the canonical map $\|A\| \to \neg\neg A$ for an arbitrary type A described in section 9.3.4:

```
‖‖-¬¬ : ∀ {i} {A : Type i} → ‖ A ‖ → ¬ (¬ A)
‖‖-¬¬ = ‖‖-rec ¬¬-isProp (λ x f → f x)
```

Appendix

A.1 Relations

A.1.1 Definition. Given a set A a *relation* R on A is a subset $R \subseteq A \times A$. We sometimes write $a \, R \, b$ when $(a, b) \in R$. It is

- *reflexive* if $a \, R \, a$ for every $a \in A$,

- *transitive* if $a \, R \, c$ for every $a, c \in A$ such that there exists $b \in A$ for which $a \, R \, b$ and $b \, R \, c$,

- *symmetric* if $b \, R \, a$ for every $a, b \in A$ such that $a \, R \, b$,

- *antisymmetric* if $a \, R \, b$ and $b \, R \, a$ implies $a = b$.

A *preorder* is a reflexive and transitive relation. A *partial order* is a reflexive, transitive and antisymmetric relation. An *equivalence relation* is a relation which is reflexive, transitive and symmetric.

A.1.2 Closure. We suppose fixed a relation R on A. Its reflexive (resp. transitive, resp. symmetric) closure is the smallest reflexive (resp. ...) relation containing R. It always exists since it can be shown to be the intersection of all reflexive (resp. ...) relations containing R. Concretely,

- the reflexive closure of R is

$$R \cup \{(a, a) \mid a \in A\}$$

- the transitive closure of R is

$$R \cup \{(a_0, a_n) \mid n > 0, (a_0, a_1) \in R, (a_1, a_2) \in R, \ldots, (a_{n-1}, a_n) \in R\}$$

- the symmetric closure of R is

$$R \cup \{(b, a) \mid (a, b) \in R\}$$

The following characterization is often useful (and similar results hold for other closure operations):

Lemma A.1.2.1. The reflexive and transitive closure R^* of a relation R on a set A is the smallest subset of A such that

- $a \, R \, a$ for every $a \in A$,

- $a \, R \, c$ for every $a, c \in A$ such that there exists $b \in A$ for which $a \, R \, b$ and $b \, R^* \, c$.

A.1.3 Quotient. An *equivalence class* E under R is a subset $E \subseteq A$ such that for every $a \in A$ and $b \in E$ such that $(a, b) \in R$, we have $a \in E$. The *quotient* A/R of A under R is the set of equivalence classes of A.

A.1.4 Congruence. Given a function $f : A^n \to A$ for some $n \in \mathbb{N}$, the relation R is a *congruence* for f when, given $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ such that $a_i \, R \, b_i$ for every $1 \leqslant i \leqslant n$, we have $f(a_1, \ldots, a_n) \, R \, f(b_1, \ldots, b_n)$. In this case, f induce a *quotient function* on A/R defined by

$$f(E_1, \ldots, E_n) = f(a_1, \ldots, a_n)$$

for some $(a_1, \ldots, a_n) \in E_1 \times \ldots \times E_n$: this function can be shown not to depend on the choice of $(a_1, \ldots, a_n)$.

A.2 Monoids

A.2.1 Definition. A *monoid* $(M, \cdot, 1)$ is a set M equipped with

- a function $\_ \cdot \_ : M \times M \to M$ called *multiplication*,

- an element $1 \in M$ called *unit*,

such that for every elements $u, v, w \in M$ we have

$$(u \cdot v) \cdot w = u \cdot (v \cdot w) \qquad\qquad 1 \cdot u = u = u \cdot 1$$

Such a monoid is

- *commutative* when $u \cdot v = v \cdot u$ for every $u, v \in M$,

- *idempotent* when $u \cdot u = u$ for every $u \in M$.

A *morphism* f from a monoid $(M, \cdot_M, 1_M)$ to a monoid $(N, \cdot_N, 1_N)$ is a function $f : M \to N$ such that

$$f(u \cdot_M v) = f(u) \cdot_N f(v) \qquad\qquad f(1_M) = f(1_N)$$

A.2.2 Free monoids. Given a set A, we write $(A^*, \cdot, 1)$ for the monoid such that A^* is the set of *words* on A, i.e. finite sequences $a_1 \ldots a_n$ of elements of A, multiplication is concatenation, i.e.

$$(a_1 \ldots a_n) \cdot (b_1 \ldots b_m) = a_1 \ldots a_n b_1 \ldots b_m$$

and unit 1 is the empty sequence. We write $|a_1 \ldots a_n| = n$ for the *length* of a word.

Proposition A.2.2.1. The monoid $(A^*, \cdot, 1)$ is the *free monoid* on A: given a monoid $(M, \cdot, 1)$ and a function $f : A \to M$, there exists a unique morphism of monoids $\overline{f}$ such that $\overline{f}(a) = f(a)$ for every $a \in A$.

Given a set A, we define in appendix A.3.5 below the set $A^{\#}$ of all multisets on A. It is a monoid when equipped with disjoint union $\uplus$ as multiplication and empty multiset $\emptyset$ as unit.

Proposition A.2.2.2. The monoid $(A^{\#}, \uplus, \emptyset)$ is the free commutative monoid on A: given a commutative monoid $(M, \cdot, 1)$ and a function $f : A \to M$, there exists a unique morphism of monoids $\overline{f}$ such that $\overline{f}(a) = f(a)$ for every $a \in A$.

Given a set A, we write $\mathcal{P}(A)$ for the set of subsets of A. It is a monoid when equipped with union $\cup$ as multiplication and empty set $\emptyset$ as unit.

Proposition A.2.2.3. The monoid $(\mathcal{P}(A), \cup, \emptyset)$ is the free idempotent commutative monoid on A: given an idempotent commutative monoid $(M, \cdot, 1)$ and a function $f : A \to M$, there exists a unique morphism of monoids $\overline{f}$ such that $\overline{f}(a) = f(a)$ for every $a \in A$.

A.3 Well-founded orders

A.3.1 Partial orders. A *partially ordered set* or *poset* $(A, \leqslant)$ is a set A equipped with a relation $\leqslant$, called *partial order* which is reflexive, transitive and antisymmetric (see also appendix A.1). A partial order is *total* when for every $a, b \in A$ we have either $a \leqslant b$ or $b \leqslant a$.

A.3.2 Well-founded orders. A poset is *well-founded* when there is no strictly decreasing infinite sequence

$$a_0 > a_1 > a_2 > \ldots$$

This is equivalent to requiring that every infinite weakly decreasing sequence

$$a_0 \geqslant a_1 \geqslant a_2 \geqslant \ldots$$

is eventually stationary

$$\exists n \in \mathbb{N}. \forall i \in \mathbb{N}. (i \geqslant n) \Rightarrow (a_i = a_{i+1})$$

A *chain* in A is a totally ordered subset of A. It is *ascending* when it has a minimal element and *descending* when it has a maximal element. A well-founded poset is thus a poset in which every descending chain is finite.

Well-founded orders are particularly interesting because they satisfy the following induction principle:

Theorem A.3.2.1 (Well-founded induction). Suppose given a property $P(a)$ on the elements a of a well-founded poset $(A, \leqslant)$. Suppose moreover that for every element $a \in A$, if $P(b)$ holds for every element $b < a$ then $P(a)$ holds. Then $P(a)$ holds for every element a of A.

Proof. By contradiction, suppose that there exists an element $a_0 \in A$ such that $P(a_0)$ does not hold. By hypothesis, this means that there is an element $a_1 < a_0$ such that $P(a_1)$ does not hold. By the same reasoning applied to a_1, we can construct an element $a_2 < a_1$ such that $P(a_2)$ does not hold. Iterating this reasoning, we construct an infinite sequence

$$a_0 > a_1 > a_2 > \ldots$$

of elements a_i such that $a_i > a_{i+1}$ and $P(a_i)$ does not hold. Since $(A, \leqslant)$ is well-founded, such a sequence cannot exist. $\qquad\square$

Remark A.3.2.2. The above proof does not exploit the fact that $<$ is transitive nor antisymmetric, and the reasoning would in fact hold for any relation R in place of $<$. A relation R on a set A is *well-founded* if there is no infinite sequence of elements a_i of A such that

$$a_0 \; R \; a_1 \; R \; a_2 \; R \ldots$$

An induction principle similar to theorem A.3.2.1 holds for such relations.

The prototypical well-founded order is the subterm order. Suppose fixed a signature Σ, see section 5.1.1. We define the *subterm order* $\leqslant$ on the terms in this signature by $u \leqslant t$ whenever u is a subterm of t, see section 5.1.2.

Lemma A.3.2.3. The relation $\leqslant$ is a partial order.

Theorem A.3.2.4. The relation $\leqslant$ is well-founded.

Proof. We define the *height* $\mathrm{ht}(t)$ of a term t by induction on t by

$$\mathrm{ht}(x) = 0 \qquad\qquad \mathrm{ht}(f(t_1, \ldots, t_n)) = 1 + \bigvee_{1 \leqslant i \leqslant n} \mathrm{ht}(t_i)$$

It is easily shown that $u < t$ implies $\mathrm{ht}(u) < \mathrm{ht}(t)$. Therefore, if the subterm order was not well-founded, $(\mathbb{N}, \leqslant)$ would not be well-founded either. $\qquad\square$

A.3.3 Lexicographic order. Given two posets $(A, \leqslant_A)$ and $(B, \leqslant_B)$, we define the *lexicographic order* $\leqslant$ on $A \times B$ by $(a, b) < (a', b')$ whenever $a < a'$, or $a = a'$ and $b < b'$.

Lemma A.3.3.1. The relation $\leqslant$ on $A \times B$ is a partial order.

Lemma A.3.3.2. The partial order $\leqslant$ is total when both $\leqslant_A$ and $\leqslant_B$ are.

Theorem A.3.3.3. The partial order $\leqslant$ is well-founded when both $\leqslant_A$ and $\leqslant_B$ are.

Proof. Suppose given an infinite sequence

$$(a_0, b_0) > (a_1, b_1) > (a_2, b_2) > \ldots$$

By definition of $>$, for every index i, we either have $a_i > a_{i+1}$ or $b_i > b_{i+1}$. The sets

$$\{i \in \mathbb{N} \mid a_i > a_{i+1}\} \qquad \text{and} \qquad \{i \in \mathbb{N} \mid b_i > b_{i+1}\}$$

are such that their union is $\mathbb{N}$, therefore one of them must be infinite. We thus have an infinite strictly decreasing sequence of elements of A or of elements of B. This is impossible since both posets $(A, \leqslant_A)$ and $(B, \leqslant_B)$ are supposed to be well-founded. $\qquad\square$

Given a well-founded poset $(A, \leqslant)$, the lexicographic order is a well-founded order on $A^2 = A \times A$, and we can iterate the construction in order to obtain a well-founded order on A^n, still called *lexicographic* and written $\leqslant_{\mathrm{lex}}$, for any natural number n, using the fact that $A^{n+1} = A \times A^n$. Finally, we can construct an order $\leqslant$ on A^*, called the *deglex order*, such that $u \leqslant v$ when

– $|u| < |v|$ (u is shorter than v), or

– $|u| = |v|$ and $u \leqslant_{\text{lex}} v$ (u is lexicographically smaller than v).

Theorem A.3.3.4. Given a well-founded poset $(A, \leqslant)$, the associated deglex order on A^* is well-founded. Moreover, it is total if the order $\leqslant$ is.

Remark A.3.3.5. This order is different from the usual dictionary order (we compare the first letter of the words, then the second, and so on) which is not well-founded: with elements $a, b \in A$ such that $a > b$, we have the infinite decreasing sequence

$$a > ba > bba > bbba > bbbba > \ldots$$

A.3.4 Trees. A (non-planar rooted) *tree* is a set T equipped with a distinguished element x_0 and a function $\tau : T \setminus \{x_0\} \to T$, satisfying

$$\forall x \in T. \exists n \in \mathbb{N}. \tau^n(x) = x_0$$

The elements of T are called the *nodes* of the tree and x_0 is called the *root* node. Given $x \in T \setminus \{x_0\}$, $\tau(x)$ is called the *parent* of x, and x is a *child* of $\tau(x)$. A node x such that $\tau^{-1}(x) = \emptyset$ is called a *leaf*. Given a node $x \in T$, the *subtree* at x is the tree

$$T_x = \{y \in T \mid \exists n \in \mathbb{N}. \tau^n(y) = x\}$$

with parent function τ_x such that $\tau_x(y) = \tau(y)$ for $y \neq x$.

Lemma A.3.4.1. The set of nodes of a tree T satisfies

$$T = \{x_0\} \cup \bigcup_{x \in \tau^{-1}(x_0)} T_x$$

where x_0 is the root of T.

A tree is *finite* when its set of nodes is finite and *infinite* otherwise. A tree T is *finitely branching* when for every node $x \in T$ its set of children $\tau^{-1}(x)$ is finite. A *branch* of a tree is a sequence of nodes $x_0, x_1, \ldots$ (finite or not) such that x_0 is the root of the tree and for every index $i > 0$, $\tau(x_i) = x_{i-1}$.

Lemma A.3.4.2 (Kőnig's lemma). A finitely-branching infinite tree has an infinite branch.

Proof. Suppose fixed a finitely-branching infinite tree T. We define an infinite branch $(x_i)_{i \in \mathbb{N}}$, with the property that the subtree at x_i is infinite, by induction on i. We set x_0 to be the root of T and, supposing that x_i is defined, we define x_{i+1} as follows. By hypothesis, the set $\tau^{-1}(x_i)$ is finite and T is infinite. From lemma A.3.4.1, we deduce that there exists $x_{i+1} \in \tau^{-1}(x)$ such that the subtree at x_{i+1} is infinite. $\square$

A *labeled* tree is a tree equipped with a function which to every node associates a *label*, which is an element of some fixed set.

A.3.5 Multisets. Suppose fixed a set A. A *multiset* is a function

$$M : A \to \mathbb{N}$$

It can be thought of as a finite collection of elements of A where each element $a \in A$ occurs $M(a)$ times. We thus write $a \in M$ whenever $M(a) > 0$. The set of multisets on A is written $A^{\#}$.

The *domain* $\mathrm{dom}(M)$ of a multiset M is the set

$$\mathrm{dom}(M) = \{a \in A \mid M(a) > 0\}$$

A multiset M is *finite* when $\mathrm{dom}(M)$ is finite. We write $A_{\mathrm{fin}}^{\#}$ for the set of finite multisets over A. We write $\emptyset$ for the *empty multiset*, such that $\emptyset(a) = 0$ for every $a \in A$. Given $a \in A$, we write $\{a\}$ for the *singleton* at a, which is the multiset such that

$$\{a\}(b) = \begin{cases} 1 & \text{if } b = a \\ 0 & \text{if } b \neq a. \end{cases}$$

Given multisets M and N on A their *union* $M \uplus N$ is defined, for $a \in A$, by

$$(M \uplus N)(a) = M(a) + N(a)$$

We write $M \subseteq N$ whenever $M(a) \leqslant N(a)$ for every $a \in A$ and in this case, we define their difference $M \setminus N$ by

$$(M \setminus N)(a) = M(a) - N(a)$$

for $a \in A$.

Suppose that $(A, \leqslant)$ is a poset. We define a partial order $\leqslant^{\#}$ on $A^{\#}$, called the *multiset extension* of $\leqslant$, by $M \leqslant^{\#} N$ whenever there exists finite multisets $X, Y \in A^{\#}$ such that

$$M = (N \setminus X) \uplus Y \qquad \text{and} \qquad \forall y \in Y. \exists x \in X. y < x$$

This order is such that we get a smaller multiset by removing and element and replacing it with an arbitrary number of smaller elements: the elements get smaller and smaller, but also more and more numerous. It can still be shown that the resulting order is well-founded when the original one is [DM79].

Theorem A.3.5.1. The poset $(A_{\mathrm{fin}}^{\#}, \leqslant^{\#})$ is well-founded if and only if $(A, \leqslant)$ is well-founded.

Proof. The left-to-right implication is easy, we show the right-to-left implication. We define a relation $\lhd$ on $A^{\#}$ by $M \lhd N$ when there exists $a \in A$ and a finite multiset Y such that

$$M = (N \setminus \{a\}) \uplus Y$$

and $b < a$ for every $b \in Y$. The relation $\leqslant^{\#}$ is easily shown to be the reflexive and transitive closure of $\lhd$. Now, by contradiction, suppose that there is an infinite decreasing sequence for $\leqslant^{\#}$. This means that there exists an infinite sequence

$$M_0 \rhd M_1 \rhd M_2 \rhd \dots$$

where

$$M_{i+1} = (M_i \setminus \{x_i\}) \uplus Y_i$$

for every index i. We construct a growing sequence of trees T_i labeled in $A \sqcup \{\bot\}$ as follows. T_0 is consisting of a root and for every element a of A, we add to the root as many sons labeled by a as the multiplicity of a in M. The tree T_{i+1} is obtained from T_i by picking an element labeled by x_i and adding to it as sons the elements of Y_i counted with multiplicities. In the case where Y_i is empty we add a single node labeled by $\bot$. The inductive limit of this process is a tree T_∞, which is infinite because at least one node is added at each step (thus the special case when Y_i is empty). We deduce from lemma A.3.4.2 that the tree T admits an infinite branch: the labels of its vertices $x_0, x_1, x_2, \ldots$ form an infinite strictly decreasing sequence of elements of A. Contradiction. $\qquad\square$

A.4 Cantor's diagonal argument

Cantor's diagonal argument is a general method to show that two sets are not in bijection. For instance, suppose that we have a bijection between sequences of elements of $\mathbb{N}$ and $\mathbb{N}$. By the bijection, we have an enumeration of all the sequences and we write $(n_i^j)_{i \in \mathbb{N}}$ for the j-th sequence. We can build a table whose columns and rows respectively correspond to i and j, and cells contain the n_i^j:

	0	1	2	3	...
0	n_0^0	n_1^0	n_2^0	n_3^0	...
1	n_0^1	n_1^1	n_2^1	n_3^1	...
2	n_0^2	n_1^2	n_2^2	n_3^2	...
3	n_0^3	n_1^3	n_2^3	n_3^3	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Then pick any sequence (m_i) such that, for every index i, m_i is a natural number different from n_i^i. Since we have an enumeration of all sequences, there is an index k such that $(m_i) = (n_i^k)$. But we have $m_k \neq n_k^k$. Contradiction. There is thus no bijection between $\mathbb{N}^\mathbb{N}$ and $\mathbb{N}$.

A.4.1 A general Cantor argument. A more general form of the Cantor argument is the following.

Theorem A.4.1.1. Suppose given sets A and B such that B contains at least two distinct elements y_0 and y_1. Then there is no surjection from A to $A \to B$.

Proof. Suppose given a surjection $\phi : A \to (A \to B)$. We consider the function $f : A \to B$ defined by

$$f(x) = \begin{cases} y_1 & \text{if } \phi(x)(x) = y_0, \\ y_0 & \text{otherwise.} \end{cases}$$

Given an element $x \in A$, we have $\phi(x)(x) \neq f(x)$ and thus ϕ is not surjective. Contradiction. $\qquad\square$

A formalization of the above proof is given below. From a constructive point of view, it requires to be able to decide equality with y_0 in B. This is of course the case when B has decidable equality, e.g. $B = \mathbb{N}$, see section 6.6.8.

Theorem A.4.1.2. Given sets A and B such that B contains at least two distinct elements y_0 and y_1, there is no injection from $A \to B$ to A.

Proof. Suppose given an injection $\psi : (A \to B) \to A$. We define a function $\phi : A \to (A \to B)$ by

$$\phi(x) = \begin{cases} x \mapsto y_0 & \text{if there is no } f : A \to B \text{ such that } \psi(f) = x, \\ f & \text{for some } f : A \to B \text{ such that } \psi(f) = x, \text{ otherwise.} \end{cases}$$

Given $f : A \to B$, we have, by definition,

$$\psi \circ \phi \circ \psi(f) = \psi(f)$$

Thus, by injectivity of ψ,

$$\phi(\psi(f)) = f$$

and ϕ is surjective. We conclude using theorem A.4.1.2. $\square$

Note that the above proof implicitly requires the excluded middle in order to construct the function ϕ. It is apparently not possible to prove this theorem in a constructive setting [Bau11].

Corollary A.4.1.3. Given a set A, write $\mathcal{P}(A)$ for its powerset. There is no surjection $A \to \mathcal{P}(A)$ and no injection $\mathcal{P}(A) \to A$. In particular, there is no bijection between A and $\mathcal{P}(A)$.

Proof. Taking $B = \{0, 1\}$ in the previous theorems, we have $\mathcal{P}(A) \simeq (A \to B)$ and we conclude. $\square$

Corollary A.4.1.4. There is no bijection between $\mathbb{N} \to \mathbb{N}$ and $\mathbb{N}$.

Proof. Take $A = B = \mathbb{N}$ in the previous theorems. $\square$

Lemma A.4.1.5. The set $\mathcal{P}$ of programs (in any reasonable language) is countable.

Proof. A program is a finite sequence of characters: writing Σ for the finite set of characters (e.g. the UTF-8 characters), programs are elements of Σ^*. In other words, writing $\mathcal{P}$ for the set of programs, we have $\mathcal{P} \subseteq \Sigma^*$. The set Σ can be totally ordered (e.g. $a < b < c < \ldots$), thus Σ^* is totally ordered by the deglex order (theorem A.3.3.4) and thus $\mathcal{P}$ is totally ordered, as a subset of a totally ordered set. Given a program $p \in \mathcal{P} \subseteq \Sigma^*$, writing n for its length, the elements below it belong to the finite set $\bigsqcup_{i \leqslant n} \Sigma^i$ which is finite, as a finite union of finite sets. We can thus associate, to every program $p \in \mathcal{P}$, the natural number n_p defined as the cardinal of the longest ascending chain in $\mathcal{P}$ with p as maximal element (which is finite by the previous argument). The function $\mathcal{P} \to \mathbb{N}$ thus defined is easily seen to be a bijection. $\square$

Corollary A.4.1.6. There is a function $\mathbb{N} \to \mathbb{N}$ which is not computable by a program.

Proof. By contradiction, suppose that this is not the case. This means that there is a surjection $\phi : \mathcal{P} \to (\mathbb{N} \to \mathbb{N})$ and, by precomposing with the isomorphism $\mathbb{N} \simeq \mathcal{P}$ of lemma A.4.1.5, a surjection $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$. We conclude by theorem A.4.1.1. $\square$

A.4.2 Agda formalization. We now provide a formalization of the above theorem A.4.1.1. Given a function $f : A \to B$ and an element y of B, the *fiber* of f at y, also called the *preimage* of y under f, is the collection of elements of A whose image is y:

```
fib : ∀ {i} {A B : Set i} → (f : A → B) → (y : B) → Set i
fib {_} {A} f y = Σ A (λ x → f x ≡ y)
```

Such a function is *surjective* when every element of B admits a pre-image under f, i.e. there exists an element in the fiber of any point:

```
surjective : ∀ {i} {A B : Set i} (f : A → B) → Set i
surjective f = ∀ y → fib f y
```

The formal proof of the theorem then follows directly from the above one. We suppose given two types A and B, the former containing two distinct elements y_0 and y_1 such that we can decide the equality to y_0, and a surjective function φ of type A → A → B and reach an absurdity:

```
no-surjection : ∀ {i} {A B : Set i} {y₀ y₁ : B} → y₀ ≢ y₁ →
                ((y : B) → Dec (y ≡ y₀)) →
                (φ : A → A → B) → surjective φ → ⊥
no-surjection {_} {A} {B} {y₀} {y₁} y₀≢y₁ dec φ surj =
  φxx≢fx x (cong-app p x)
  where
  f : A → B
  f x with dec (φ x x)
  f x | yes _ = y₁
  f x | no  _ = y₀
  φxx≢fx : (x : A) → φ x x ≢ f x
  φxx≢fx x p with dec (φ x x)
  φxx≢fx x p | yes refl = y₀≢y₁ p
  φxx≢fx x p | no ¬p    = ¬p p
  x : A
  x = fst (surj f)
  p : φ x ≡ f
  p = snd (surj f)
```

Note that the construction of f requires us to be able to decide equality with y_0 which we also have to suppose given as argument. The proof of theorem A.4.1.2 can also be formalized if we assume the law of excluded middle, called `lem` below. We define the predicate of being injective by

```
injective : ∀ {i} {A B : Set i} (f : A → B) → Set i
injective f = ∀ {x x'} → f x ≡ f x' → x ≡ x'
```

and then show the theorem by following the proof given above, which is based on the previous function

```
no-injection : ∀ {i} {A B : Set i} {y₀ y₁ : B} → y₀ ≢ y₁ →
               (lem : (A : Set i) → Dec A) →
               (ψ : (A → B) → A) → injective ψ → ⊥
no-injection {_} {A} {B} {y₀} {y₁} y₀≢y₁ lem ψ inj =
```

```
no-surjection y₀≠y₁ (λ y → lem (y ≡ y₀)) φ surj
where
φ : A → A → B
φ x with lem (fib ψ x)
φ x | yes (f , p) = f
φ x | no ¬p = λ _ → y₀
ψφψ≡ψ : (f : A → B) → ψ (φ (ψ f)) ≡ ψ f
ψφψ≡ψ f with lem (fib ψ (ψ f))
ψφψ≡ψ f | yes (g , p) = p
ψφψ≡ψ f | no ¬p = ⊥-elim (¬p (f , refl))
surj : surjective φ
surj f = ψ f , inj (ψφψ≡ψ f)
```

Bibliography

[Abe17]   Andreas Abel. How safe is Type:Type? Mail on the Agda mailing list, 2017. Available at `https://lists.chalmers.se/pipermail/agda/2017/009337.html`.

[ACD+18]   Andreas Abel, Jesper Cockx, Dominique Devriese, Amin Timany, and Philip Wadler. $\doteq\simeq\cong$ Leibniz Equality is Isomorphic to Martin-Löf Identity, Parametrically. Unpublished, 2018.

[Ack28]   Wilhelm Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99(1):118–133, 1928.

[Acz78]   Peter Aczel. The Type Theoretic Interpretation of Constructive Set Theory. In *Studies in Logic and the Foundations of Mathematics*, volume 96, pages 55–66. Elsevier, 1978.

[Alt19]   Thorsten Altenkirch. Naïve type theory. In *Reflections on the Foundations of Mathematics*, pages 101–136. Springer, 2019.

[Arn17]   Michael Arntzenius. Normalisation by evaluation for the simply-typed lambda calculus, in Agda, 2017. Available at `https://gist.github.com/rntz/2543cf9ef5ee4e3d990ce3485a0186e2/revisions`.

[Bae18]   John Baez. Patterns That Eventually Fail. Azimuth blog, 2018. `https://johncarlosbaez.wordpress.com/2018/09/20/patterns-that-eventually-fail/`.

[Bar84]   Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[Bar91]   Henk Barendregt. Self-interpretation in lambda calculus. *Journal of Functional Programming*, 1(2):229–233, 1991.

[Bau11]   Andrej Bauer. An injection from $\mathbb{N}^{\mathbb{N}}$ to $\mathbb{N}$. Unpublished note, 2011.

[Bau12]   Andrej Bauer. How to implement dependent type theory. Blog post, 2012. Available at `http://math.andrej.com/2012/11/08/how-to-implement-dependent-type-theory-i/`.

[Bau17]   Andrej Bauer. Five stages of accepting constructive mathematics. *Bulletin of the American Mathematical Society*, 54(3):481–498, 2017.

[BB01]   David Borwein and Jonathan M Borwein. Some remarkable properties of sinc and related integrals. *The Ramanujan Journal*, 5(1):73–89, 2001.

[Bel98]   John Lane Bell. *A primer of infinitesimal analysis.* Cambridge University Press, 1998.

[BF97]   Cesare Burali-Forti. Una questione sui numeri transfiniti. *Rendiconti del Circolo Matematico di Palermo (1884-1940)*, 11(1):154–164, 1897.

[Bla84]   Andreas Blass. Existence of bases implies the axiom of choice. *Contemporary Mathematics*, 31, 1984.

[BN99]   Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge university press, 1999.

[Boo84]   George Boolos. Don't eliminate cut. *Journal of Philosophical Logic*, pages 373–378, 1984.

[BPT17]   Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Certified Programs and Proofs (CPP 2017)*, pages 182–194, 2017.

[BW97]   Bruno Barras and Benjamin Werner. Coq in Coq. Available at `http://www.lix.polytechnique.fr/~barras/publi/coqincoq.pdf`, 1997.

[CCHM16]   Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *arXiv preprint arXiv:1611.02108*, 2016.

[CF58]   Haskell B Curry and Robert Feys. Combinatory logic. *Studies in Logic and the Foundations of Mathematics*, 1, 1958.

[CH00]   Pierre-Louis Curien and Hugo Herbelin. The duality of computation. *ACM sigplan notices*, 35(9):233–243, 2000.

[Cha12]   Arthur Charguéraud. The Locally Nameless Representation. *Journal of automated reasoning*, 49(3):363–408, 2012.

[Chu36a]   Alonzo Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, (1):40–41, 1936.

[Chu36b]   Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, (58):345–363, 1936.

[Chu40]   Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[CK90]   Chen Chung Chang and H Jerome Keisler. *Model theory.* Elsevier, 1990.

[CKA15]   Leran Cai, Ambrus Kaposi, and Thorsten Altenkirch. Formalising the Completeness Theorem of Classical Propositional Logic in Agda (Proof Pearl). Unpublished manuscript, 2015.

[CKNT09]   Thierry Coquand, Yoshiki Kinoshita, Bengt Nordström, and Makoto Takeyama. A simple type-theoretic language: Mini-TT. *From Semantics to Computer Science; Essays in Honour of Gilles Kahn*, pages 139–164, 2009.

[CL93]   René Cori and Daniel Lascar. *Logique mathématique: cours et exercices. Calcul propositionnel, algèbres de Boole, calcul des prédicats.* Masson, 1993.

[CLRS09]   Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2009.

[CMP00]   Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml.* O'Reilly Sebastopol, CA, 2000.

[Coq86]   Thierry Coquand. An analysis of Girard's paradox. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86)*, pages 227–236. IEEE Computer Society, 1986.

[Coq92a]   Thierry Coquand. The paradox of trees in type theory. *BIT Numerical Mathematics*, 32(1):10–14, 1992.

[Coq92b]   Thierry Coquand. Pattern matching with dependent types. In *Informal proceedings of Logical Frameworks*, volume 92, pages 66–79, 1992.

[Coq95]   Thierry Coquand. A new paradox in type theory. In *Studies in Logic and the Foundations of Mathematics*, volume 134, pages 555–570. Elsevier, 1995.

[Coq96]   Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, 1996.

[Coq13]   Thierry Coquand. Defining coinductive types. Mail on the Agda mailing list, December 2013. `https://lists.chalmers.se/pipermail/agda/2013/006189.html`.

[CP88]   Thierry Coquand and Christine Paulin. Inductively defined types. In *International Conference on Computer Logic*, pages 50–66. Springer, 1988.

[CR36]   Alonzo Church and J Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[Cur30]   Haskell Brooks Curry. Grundlagen der kombinatorischen logik. *American journal of mathematics*, 52(4):789–834, 1930.

[Deh17]   Patrick Dehornoy. *Théorie des ensembles: Introduction à une théorie de l'infini et des grands cardinaux.* Calvage et Mounet, 2017.

[dGdBB+19] Stijn de Gouw, Frank S de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. Verifying openjdk's sort method for generic collections. *Journal of automated reasoning*, 62(1):93–126, 2019.

[Dia75] Radu Diaconescu. Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, 51(1):176–178, 1975.

[Dij70] Edsger Wybe Dijkstra. Notes on Structured Programming, 1970.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.

[DMJ16] Hannes Diener and Maarten McKubre-Jordens. Classifying Material Implications over Minimal Logic. *arXiv preprint arXiv:1606.08092*, 2016.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[Dum59] Michael Dummett. A propositional calculus with denumerable matrix. *The Journal of Symbolic Logic*, 24(2):97–106, 1959.

[Dyb94] Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.

[Dyc92] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992.

[Esc19] Martín Hötzel Escardó. Introduction to Univalent Foundations of Mathematics with Agda. Course notes, 2019. Available at `https://www.cs.bham.ac.uk/~mhe/HoTT-UF-in-Agda-Lecture-Notes/`.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, 103(2):235–271, 1992.

[FR98] Michael J Fischer and Michael O Rabin. Super-exponential complexity of Presburger arithmetic. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 122–135. Springer, 1998.

[Fre79] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Nebert, 1879.

[FS12]   Fredrik Nordvall Forsberg and Anton Setzer. A finite axioma-
           tisation of inductive-inductive definitions. *Logic, Construction,
           Computation*, 3:259–287, 2012.

[GAA+13]   Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot,
           Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mah-
           boubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-
           checked proof of the odd order theorem. In *International Con-
           ference on Interactive Theorem Proving*, pages 163–179. Springer,
           2013.

[Gal89]   Jean H Gallier. *Logic and Computer Science*, chapter On Girard's
           "Candidats de Reductibilité". Academic Press, 1989.

[Gen35]   Gerhard Gentzen. Untersuchungen über das logische Schließen.
           Mathematische zeitschrift, 39(1):176–210, 405–431, 1935.

[Gen36]   Gerhard Gentzen. Die Widerspruchsfreiheit der reinen Zahlenthe-
           orie. *Mathematische Annalen*, 112(1):493–565, 1936.

[Gir72]   Jean-Yves Girard. *Interprétation fonctionnelle et élimination des
           coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Univer-
           sité Paris Diderot - Paris 7, 1972.

[Gir87]   Jean-Yves Girard. Linear logic. *Theoretical computer science*,
           50(1):1–101, 1987.

[Gir89]   Jean-Yves Girard. *Proofs and types*, volume 7. Cambridge univer-
           sity press Cambridge, 1989.

[Gir11]   Jean-Yves Girard. *The Blind Spot: lectures on logic*. European
           Mathematical Society, 2011.

[GL02]   Benjamin Grégoire and Xavier Leroy. A compiled implementa-
           tion of strong reduction. *ACM SIGPLAN Notices*, 37(9):235–246,
           2002.

[Gli29]   Valery Glivenko. Sur quelques points de la logique de M. Brouwer.
           Bulletins de la classe des sciences, 15(5):183–188, 1929.

[GLW99]   Didier Galmiche and Dominique Larchey-Wendling. Structural
           sharing and efficient proof-search in propositional intuitionistic
           logic. In *Annual Asian Computing Science Conference*, pages 101–
           112. Springer, 1999.

[GM78]   Nelson Goodman and John Myhill. Choice implies excluded mid-
           dle. *Mathematical Logic Quarterly*, 24(25-30):461–461, 1978.

[Göd31]   Kurt Gödel. Über formal unentscheidbare Sätze der Principia
           Mathematica und verwandter Systeme I. *Monatshefte für mathe-
           matik und physik*, 38(1):173–198, 1931.

[God32]   Kurt Godel. Zum intuitionistischen Aussagenkalkul.
           *Anzeiger Akademie der Wissenschaften Wien, mathematisch-
           naturwissenschaftliche Klasse*, 69:65–66, 1932.

[Göd38]   Kurt Gödel. The consistency of the axiom of choice and of the generalized continuum-hypothesis. *Proceedings of the National Academy of Sciences of the United States of America*, 24(12):556, 1938.

[Göd58]   Von Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12(3-4):280–287, 1958.

[Gon08]   Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[Gri89]   Timothy G Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–58, 1989.

[HA28]   David Hilbert and Wilhelm Ackermann. Grundzüge der theoretischen logik. 1928.

[HAB+17]   Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof of the kepler conjecture. In *Forum of mathematics, Pi*, volume 5. Cambridge University Press, 2017.

[Har11]   Robert Harper. Computational trinitarianism, 2011. See `https://ncatlab.org/nlab/show/computational+trinitarianism`.

[Hat02]   Allen Hatcher. *Algebraic topology*. Cambridge University Press, Cambridge, 2002.

[Hed98]   Michael Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998.

[Her30]   Jacques Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, Faculté des sciences de Paris, 1930.

[Hil22]   David Hilbert. Neubegründung der Mathematik (erste Mitteilung). *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 1(1):157–177, 1922.

[Hin69]   Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[Hof95]   Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, University of Edinburgh, 1995.

[How80]   William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[HS98]   Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. *Twenty-five years of constructive type theory (Venice, 1995)*, 36:83–111, 1998.

[Hue94]   Gérard Huet. Residual theory in λ-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.

[Hur95]   Antonius JC Hurkens. A simplification of Girard's paradox. In *International Conference on Typed Lambda Calculi and Applications*, pages 266–278. Springer, 1995.

[Hur10]   Chung Kil Hur. Agda with excluded middle is inconsistent. E-mail, 2010. Available at `https://lists.chalmers.se/pipermail/agda/2010/001526.html`.

[Jac99]   Bart Jacobs. *Categorical logic and type theory*. Elsevier, 1999.

[KECA16]   Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of anonymous existence in Martin-Löf type theory. *Logical Methods in Computer Science*, 2016.

[Kis13]   Oleg Kiselyov. How OCaml type checker works – or what polymorphism and garbage collection have in common, 2013. `http://okmij.org/ftp/ML/generalization.html`.

[KL20]   Chris Kapulkin and Peter LeFanu Lumsdaine. The law of excluded middle in the simplicial model of type theory, 2020.

[Kna28]   Bronisław Knaster. Un théorème sur les functions d'ensembles. *Ann. Soc. Polon. Math.*, 6:133–134, 1928.

[Koc06]   Anders Kock. *Synthetic differential geometry*, volume 333. Cambridge University Press, 2006. Available at `http://home.imf.au.dk/kock/sdg99.pdf`.

[KP57]   Georg Kreisel and Hilary Putnam. Eine Unableitbarkeitsbeweismethode für den intuitionistischen Aussagenkalkül. *Archiv für mathematische Logik und Grundlagenforschung*, 3(3-4):74–78, 1957.

[KP82]   Laurie Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14(4):285–293, 1982.

[Kri65]   Saul A Kripke. Semantical analysis of intuitionistic logic I. In *Studies in Logic and the Foundations of Mathematics*, volume 40, pages 92–130. Elsevier, 1965.

[Kri98]   Jean-Louis Krivine. *Théorie des ensembles*. Cassini, 1998.

[Kri09]   Neelakantan R Krishnaswami. Focusing on pattern matching. In *POPL*, volume 9, pages 366–378, 2009.

[KV91]   Mikhail M Kapranov and Vladimir A Voevodsky. ∞-groupoids and homotopy types. *Cahiers de topologie et géométrie différentielle catégoriques*, 32(1):29–46, 1991.

[Lei86]   Gottfried Wilhelm Leibniz. *Discours de métaphysique*. 1686.

[LMS10]   Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta informaticae*, 102(2):177–207, 2010.

[LS88]   Joachim Lambek and Philip J Scott. *Introduction to higher-order categorical logic*, volume 7. Cambridge University Press, 1988.

[Lyn17]   Ben Lynn. Lambda Calculus, 2017. `https://crypto.stanford.edu/~blynn/lambda/`.

[Mac71]   Saunders MacLane. Categories for the working mathematician. *Graduate texts in mathematics*, 5, 1971.

[McB00]   Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, 2000.

[McC60]   John McCarthy. *Programs with common sense*. RLE and MIT computation center, 1960.

[Mil78]   Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[ML]   Per Martin-Löf. The collected works of Per Martin-Löf. `https://github.com/michaelt/martin-lof`.

[ML75]   Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

[ML82]   Per Martin-Löf. Constructive mathematics and computer programming. In *Studies in Logic and the Foundations of Mathematics*, volume 104, pages 153–175. Elsevier, 1982.

[ML98]   Per Martin-Löf. An intuitionistic theory of types. *Twenty-five years of constructive type theory*, 36:127–172, 1998.

[MLS84]   Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[MMH13]   Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. O'Reilly Media, Inc., 2013.

[Mog91]   Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[Mog92]   Torben Ægidius Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, 1992.

[Mun19]   Randall Munroe. Differentiation and Integration, 2019. `https://xkcd.com/2117/`.

[Nor07]   Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Göteborg university, 2007.

[Ore82]   Vladimir P Orevkov. Lower bounds for increasing complexity of derivations after cut elimination. *Journal of Soviet Mathematics*, 20(4):2337–2350, 1982.

[Par92]   Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 190–201. Springer, 1992.

[Par97]   Michel Parigot. Proofs of strong normalisation for second order classical natural deduction. *The Journal of Symbolic Logic*, 62(4):1461–1479, 1997.

[PdAC+10]   Benjamin C Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hricu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. Software Foundations 2: Programming Language Foundations, 2010. Available at https://softwarefoundations.cis.upenn.edu/plf-current/.

[Pie02]   Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[PM93]   Christine Paulin-Mohring. Inductive Definitions in the System Coq Rules and Properties. In *International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer, 1993.

[PR05]   François Pottier and Didier Rémy. The essence of ML type inference, 2005.

[Pre29]   Mojzesz Presburger. Über die Vollstandigkeiteines gewissen Systems der Arithmetik ganzer Zahlen, in welchen die Addition als einzige Operation hervortritt. In *Comptes-Rendus du 1er Congrès des Mathématiciens des Pays Slaves*, pages 92–101, 1929.

[Rém92]   Didier Rémy. Extension of ML Type System with a Sorted Equational Theory on Types. Technical Report 1766, INRIA, October 1992.

[Rob65]   John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[Sak14]   Kazuhiko Sakaguchi. Formalizing Strong Normalization Proofs. In *Theorem proving and provers for reliable theory and implementations*, volume 61, pages 16–23, 2014.

[Sch24]   Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische annalen*, 92(3):305–316, 1924.

[Sch22]   Peter Scholze. Liquid tensor experiment. *Experimental Mathematics*, 31(2):349–354, 2022.

[Sel02]   Peter Selinger. The Lambda Calculus is Algebraic. *Journal of Functional Programming*, 12(6):549–566, 2002.

[Sel08]   Peter Selinger. Lecture notes on the lambda calculus. Lecture notes, 2008.

[Sim98]   Carlos Simpson. Homotopy types of strict 3-groupoids. *arXiv preprint math/9810059*, 1998.

[Sta79]   Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9(1):67–72, 1979.

[Str93]   Thomas Streicher. Investigations into intensional type theory. *Habilitiation Thesis, Ludwig Maximilian Universität*, 1993.

[SU06]   Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.

[Tai75]   William W Tait. A realizability interpretation of the theory of species. In *Logic Colloquium*, pages 240–251. Springer, 1975.

[Tar55]   Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[Tur37a]   Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, (42):230–265, 1936-1937.

[Tur37b]   Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[Tur49]   Alan Turing. On checking a large routine. In *Report of a Conference on Speed Automatic Calculating Machines*, pages 67–69, 1949.

[Uni13]   The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[Voe14]   Vladimir Voevodsky. Univalent foundations, March 2014. Presentation at IAS, `http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2014_IAS.pdf`.

[Wer97]   Benjamin Werner. Sets in types, types in sets. In *International Symposium on Theoretical Aspects of Computer Software*, pages 530–546. Springer, 1997.

[Wie06]   Freek Wiedijk. *The Seventeen Provers of the World*, volume 3600. Springer, 2006.

[WK19]   Philip Wadler and Wen Kokke. Programming language foundations in Agda, 2019. Available at `http://plfa.inf.ed.ac.uk/`.

[WR12]   Alfred North Whitehead and Bertrand Russell. *Principia mathematica*, volume 2. University Press, 1912.

[WZ07]   Frank Wolter and Michael Zakharyaschev. Modal decision problems. In *Studies in Logic and Practical Reasoning*, volume 3, pages 427–489. Elsevier, 2007.

[Zer08]   Ernst Zermelo. Untersuchungen über die Grundlagen der Mengenlehre. I. *Mathematische Annalen*, 65(2):261–281, 1908.

Index