CSC_51051_EP: Agda

Samuel Mimram

2025

École polytechnique

Part I

Introduction

Curry-Howard on steroids

We have seen that types can be seen as formulas and programs as proofs:

`'a -> 'a * 'b` corresponds to $A \Rightarrow A \wedge B$

and this language is a subset of OCaml (λ-calculus).

We are now going to see and use **Agda**, which is a programming language in which types are much more expressive than propositional logic.

Curry-Howard on steroids

For instance, the type of division in OCaml is

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

Curry-Howard on steroids

For instance, the type of division in OCaml is

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

We are going to be able to give it a type such as

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

which can be read as the formula

$$\forall m \in \mathtt{int}.\forall n \in \mathtt{int}.\exists q \in \mathtt{int}.\exists r \in \mathtt{int}.((m = nq + r) \land (r < n))$$

Proof assistants

A **proof** assistant is a software which helps you writing **proofs**:

1. it checks that your proof is actually correct (type checking),

Proof assistants

A **proof** assistant is a software which helps you writing **proofs**:

1. it checks that your proof is actually correct (type checking),
2. it provides you tools to gradually elaborate a proof,

Proof assistants

A **proof** assistant is a software which helps you writing **proofs**:

1. it checks that your proof is actually correct (type checking),
2. it provides you tools to gradually elaborate a proof,
3. it provide tools to automate some parts of the proofs,

Proof assistants

A **proof** assistant is a software which helps you writing **proofs**:

1. it checks that your proof is actually correct (type checking),
2. it provides you tools to gradually elaborate a proof,
3. it provide tools to automate some parts of the proofs,
4. it provide you a way to execute the proofs or extract code (Curry-Howard).

Proof assistants

A **proof** assistant is a software which helps you writing **proofs**:

1. it checks that your proof is actually correct (type checking),
2. it provides you tools to gradually elaborate a proof,
3. it provide tools to automate some parts of the proofs,
4. it provide you a way to execute the proofs or extract code (Curry-Howard).

Most well-known proof assistants: Agda, Coq, Isabelle, Lean, etc.

Pros and cons

Once you prove a program, you are 100% sure that your proof is **correct** or that your program satisfies the specification.

Pros and cons

Once you prove a program, you are 100% sure that your proof is **correct** or that your program satisfies the specification.

But there is a price to pay: every case has to be handled in full details which means that it takes quite much time (and money).

Pros and cons

Once you prove a program, you are 100% sure that your proof is **correct** or that your program satisfies the specification.

But there is a price to pay: every case has to be handled in full details which means that it takes quite much time (and money).

In particular, type inference is undecidable so that we have to somehow explain how to type the term.

Safety

In theory, once you make your proof in a proof assistant you are sure that your proof is correct, excepting that:

Safety

In theory, once you make your proof in a proof assistant you are sure that your proof is correct, excepting that:

- the theory might be wrong

Safety

In theory, once you make your proof in a proof assistant you are sure that your proof is correct, excepting that:

- the theory might be wrong
- the implementation of the proof assistant might be buggy

Safety

In theory, once you make your proof in a proof assistant you are sure that your proof is correct, excepting that:

- the theory might be wrong
- the implementation of the proof assistant might be buggy
- the specification might be wrong or imprecise

Safety

In theory, once you make your proof in a proof assistant you are sure that your proof is correct, excepting that:

- the theory might be wrong
  quite unlikely
- the implementation of the proof assistant might be buggy
- the specification might be wrong or imprecise

Safety

In theory, once you make your proof in a proof assistant you are sure that your proof is correct, excepting that:

- the theory might be wrong
  quite unlikely
- the implementation of the proof assistant might be buggy
  unlikely: de Bruijn criterion + self-formalization
  e.g. `https://github.com/coq-contribs/coq-in-coq`
- the specification might be wrong or imprecise

Safety

In theory, once you make your proof in a proof assistant you are sure that your proof is correct, excepting that:

- the theory might be wrong
  quite unlikely
- the implementation of the proof assistant might be buggy
  unlikely: de Bruijn criterion + self-formalization
  e.g. `https://github.com/coq-contribs/coq-in-coq`
- the specification might be wrong or imprecise
  this does happen

Agda

Starting from now we are going to use Agda:

- we introduce the programming language,
- we explain the theory behind it (expanding on previous courses),
- in labs, you will get to the point of proving (simple) algorithms.

It might seem quite some syntax to absorb but you should get used to it with the labs.

Starting from now we are going to use Agda:

- we introduce the programming language,
- we explain the theory behind it (expanding on previous courses),
- in labs, you will get to the point of proving (simple) algorithms.

It might seem quite some syntax to absorb but you should get used to it with the labs.

We chose it because

- it is Curry-Howard in its purest form,
- it is really minimal: we define everything (e.g. product or equality) from a very restricted number of basic constructions.

Part II

A first proof

Commutativity of the product in OCaml

Recall from the first course that the formula

$$(A \land B) \Rightarrow (B \land A)$$

can be proved in λ-calculus by

Commutativity of the product in OCaml

Recall from the first course that the formula

$$(A \land B) \Rightarrow (B \land A)$$

can be proved in λ-calculus by

$$\lambda p^{A \land B}.\langle \pi_{\mathsf{r}}(p), \pi_{\mathsf{l}}(p) \rangle$$

Commutativity of the product in OCaml

Recall from the first course that the formula

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

can be proved in λ-calculus by

$$\lambda p^{A \wedge B}.\langle \pi_{\mathrm{r}}(p), \pi_{\mathrm{l}}(p) \rangle$$

and can be proved in OCaml by

Commutativity of the product in OCaml

Recall from the first course that the formula

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

can be proved in λ-calculus by

$$\lambda p^{A \wedge B}.\langle \pi_\mathsf{r}(p), \pi_\mathsf{l}(p) \rangle$$

and can be proved in OCaml by

```
# let prod_com (a , b) = (b , a);;
val prod_com : 'a * 'b -> 'b * 'a = <fun>
```

Commutativity of the product in OCaml

Recall from the first course that the formula

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

can be proved in λ-calculus by

$$\lambda p^{A \wedge B}.\langle \pi_{\mathsf{r}}(p), \pi_{\mathsf{l}}(p) \rangle$$

and can be proved in OCaml by

```
# let prod_com (a , b) = (b , a);;
val prod_com : 'a * 'b -> 'b * 'a = <fun>
```

We can do the same in Agda.

Our first proof

```
open import Data.Product

-- The product is commutative
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B (a , b) = (b , a)
```

Note: modules import, comments, utf-8 symbols, type / function definition, matching, Set, spaces, dependent types, two interpretations (Curry-Howard).

Inputting the code

In order to type Agda code you should use Emacs or VSCode with appropriate support.

Inputting the code

In order to type Agda code you should use Emacs or VSCode with appropriate support.

Agda is fond of the use of funny symbols:

- × is typed `\times` (VSCode: `*times`),
- → is typed `\to` (VSCode: `*to`),...

Inputting the code

In order to type Agda code you should use Emacs or VSCode with appropriate support.

Agda is fond of the use of funny symbols:

- × is typed `\times` (VSCode: `*times`),
- → is typed `\to` (VSCode: `*to`),...

Once you have finished typing the code, you should type

```
C-c C-l
```

(control+c then control+l) in order to have Agda

1. load our code (do it whenever you changed the file),
2. highlight your code,
3. check that it is correct.

Table of symbols

For reference, the common symbols are:

$\wedge$	`\and`	$\top$	`\top`	$\to$	`\to`	$\forall$	`\all`	Π	`\Pi`	λ	`\Gl`
$\vee$	`\or`	$\bot$	`\bot`	$\neg$	`\neg`	$\exists$	`\ex`	Σ	`\Sigma`	$\equiv$	`\equiv`

and some other useful ones are

$\mathbb{N}$	`\bN`	$\times$	`\times`	$\leqslant$	`\le`	$\in$	`\in`	$\uplus$	`\uplus`	$::$	`\::`	$\blacksquare$	`\qed`

Spaces

Agda is very picky about *spaces*: they are needed around operations.

This means that

```
x + y
```

is an addition, whereas

```
x+y
```

is an identifier.

Programs with holes

In practice, it is almost impossible to directly write a full Agda program correctly.

We generally proceed by **refinement** by writing

$$?$$

which is a **hole** meaning "I'll see later how I can fill that".

Programs with holes

In practice, it is almost impossible to directly write a full Agda program correctly.

We generally proceed by **refinement** by writing

```
?
```

which is a **hole** meaning "I'll see later how I can fill that".

For instance, in our example, we would write

```
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B p = ?
```

Shortcuts

```
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B p = ?
```

We then have shortcuts to help us in proofs:

`C-c C-l`	typecheck and highlight the current file
`C-c C-,`	get information about the hole under the cursor
`C-c C-.`	same as above + the type of the proposed filler
`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

Shortcuts

```
×-comm : (A B : Set) → (A × B) → (B × A)
×-comm A B p = ?
```

We then have shortcuts to help us in proofs:

`C-c C-l`	typecheck and highlight the current file
`C-c C-,`	get information about the hole under the cursor
`C-c C-.`	same as above + the type of the proposed filler
`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

NB: we can fill holes with expressions containing `?`

Set

In Agda everything has to have a type.

Therefore, they have introduced a type

```
Set
```

such that the values of this type are types: this is the type of all types.

(yes, this sounds wonderful and scaring at the same time)

(more on this later on)

Part III

Arrow types

Arrow types

The type for “usual” functions is

```
A → B
```

which can either be read as

- the type of functions which take an x of type A and return something of type B:

```
A -> B
```

- an implication:

$$A \Rightarrow B$$

Arrow types

For instance, we can prove

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

by

Arrow types

For instance, we can prove

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

by

```
thm : (A B : Set) → A → (A → B) → B
thm A B a f = f a
```

Implicit arguments

The arguments `A` and `B` have to be given each time, which is kind of heavy:

Implicit arguments

The arguments `A` and `B` have to be given each time, which is kind of heavy:

```
open import Data.Nat

p : ℕ × ℕ
p = ×-comm ℕ ℕ (5 , 4)
```

Implicit arguments

The arguments `A` and `B` have to be given each time, which is kind of heavy:

```
open import Data.Nat

p : ℕ × ℕ
p = ×-comm ℕ ℕ (5 , 4)
```

Fortunately, we can make them **implicit**:

Implicit arguments

The arguments `A` and `B` have to be given each time, which is kind of heavy:

```
open import Data.Nat

p : ℕ × ℕ
p = ×-comm ℕ ℕ (5 , 4)
```

Fortunately, we can make them **implicit**:

```
×-comm : {A B : Set} → (A × B) → (B × A)
×-comm (a , b) = b , a

p : ℕ × ℕ
p = ×-comm (5 , 4)
```

Implicit arguments

The arguments `A` and `B` have to be given each time, which is kind of heavy:

```
open import Data.Nat

p : ℕ × ℕ
p = ×-comm ℕ ℕ (5 , 4)
```

Fortunately, we can make them **implicit**:

```
×-comm : {A B : Set} → (A × B) → (B × A)
×-comm (a , b) = b , a

p : ℕ × ℕ
p = ×-comm (5 , 4)
```

NB: we can check the resulting value for `p` with `C-c C-n`.

Arrow types: λ

The identity can be written as:

Arrow types: λ

The identity can be written as:

```
id : {A : Set} → A → A
id a = a
```

Arrow types: λ

The identity can be written as:

```
id : {A : Set} → A → A
id a = a
```

We can also make anonymous functions:

Arrow types: λ

The identity can be written as:

```
id : {A : Set} → A → A
id a = a
```

We can also make anonymous functions:

```
id : {A : Set} → A → A
id = λ a → a
```

This is akin OCaml:

```
let id x = x
let id = fun x -> x
```

Part IV

Inductive types

Inductive types: booleans

We can define **inductive types**, e.g. booleans:

Inductive types: booleans

We can define **inductive types**, e.g. booleans:

```
data Bool : Set where
  false : Bool
  true  : Bool
```

(note: there are *2 spaces* before each constructor)

Inductive types: booleans

We can define **inductive types**, e.g. booleans:

```
data Bool : Set where
  false : Bool
  true  : Bool
```

on which we define functions by **induction**, e.g. negation:

Inductive types: booleans

We can define **inductive types**, e.g. booleans:

```
data Bool : Set where
  false : Bool
  true  : Bool
```

on which we define functions by **induction**, e.g. negation:

```
not : Bool → Bool
not false = true
not true  = false
```

Inductive types: booleans

We can define **inductive types**, e.g. booleans:

```
data Bool : Set where
  false : Bool
  true  : Bool
```

on which we define functions by **induction**, e.g. negation:

```
not : Bool → Bool
not false = true
not true  = false
```

NB: we can see that automatic fill is not a good idea!

Inductive types: booleans

We can define **inductive types**, e.g. booleans:

```
data Bool : Set where
  false : Bool
  true  : Bool
```

on which we define functions by **induction**, e.g. negation:

```
not : Bool → Bool
not false = true
not true  = false
```

NB: we can see that automatic fill is not a good idea!

In the standard library: `Data.Bool`.

Inductive types: natural numbers

Similarly, how do we define **natural numbers**?

Inductive types: natural numbers

Similarly, how do we define **natural numbers**?

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Inductive types: natural numbers

Similarly, how do we define **natural numbers**?

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

We define addition by induction by

Inductive types: natural numbers

Similarly, how do we define **natural numbers**?

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

We define addition by induction by

```
add : ℕ → ℕ → ℕ
add zero    n = n
add (suc m) n = suc (add m n)

x : ℕ
x = add (suc zero) (suc (suc zero))
```

Note that we can call recursively ourselves.

Inductive types: natural numbers

Similarly, how do we define **natural numbers**?

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Or better, we can use **infix** notation:

```
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)


x : ℕ
x = (suc zero) + (suc (suc zero))
```

Inductive types: natural numbers

Similarly, how do we define **natural numbers**?

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

And we can add more sugar for numbers:

```
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)

{-# BUILTIN NATURAL ℕ #-}

x : ℕ
x = 3 + 2
```

Inductive types: natural numbers

Similarly, how do we define **natural numbers**?

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

In the standard library: `Data.Nat`.

Inductive types: natural numbers

Similarly, how do we define **natural numbers**?

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

There is also a syntax for anonymous functions with pattern matching:

```
add : ℕ → ℕ → ℕ
add = λ { zero n → zero ; (suc m) n → suc (add m n) }
```

Inductive types: natural numbers

Similarly, how do we define **natural numbers**?

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

The inductive definition intuitively means that ℕ is the smallest set of terms such that

- `zero` belongs to ℕ,
- if n belongs to ℕ then there is a *new* term `suc` n which belongs to ℕ.

Inductive types: natural numbers

Similarly, how do we define **natural numbers**?

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

The inductive definition intuitively means that ℕ is the smallest set of terms such that

- `zero` belongs to ℕ,
- if n belongs to ℕ then there is a *new* term `suc` n which belongs to ℕ.

In particular, constructors are **injective**:

- `zero` is never the same as `suc` n,
- if `suc` m is the same as `suc` n then necessarily m is the same as n.

Termination

In Agda, all functions must terminate:

```
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc m + n
```

gives rise to the error

Termination

In Agda, all functions must terminate:

```
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc m + n
```

gives rise to the error

```
Termination checking failed for the following functions:
  _+_
Problematic calls:
  suc m + n
```

Termination

Allowing functions to be non-terminating would make the system incoherent:

Termination

Allowing functions to be non-terminating would make the system incoherent:

```
{-# TERMINATING #-}
anything : {A : Set} → ℕ → A
anything n = anything (suc n)
```

Termination

Allowing functions to be non-terminating would make the system incoherent:

```
{-# TERMINATING #-}
anything : {A : Set} → ℕ → A
anything n = anything (suc n)
```

From which we can deduce pretty much whatever we want:

```
open import Relation.Binary.PropositionalEquality

absurd : 0 ≡ 1
absurd = anything 25
```

Termination

Allowing functions to be non-terminating would make the system incoherent:

```
{-# TERMINATING #-}
anything : {A : Set} → A
anything = anything
```

From which we can deduce pretty much whatever we want:

```
open import Relation.Binary.PropositionalEquality

absurd : 0 ≡ 1
absurd = anything
```

Termination

Note that because of termination all functions are **total** in Agda: given an argument, they always produce an output.

(this is not the case in OCaml for instance)

Termination

Theorem
Agda is a programming language in which every programmable function is total, therefore there is a total computable function which cannot be implemented.

Proof.

Termination

Theorem
Agda is a programming language in which every programmable function is total, therefore there is a total computable function which cannot be implemented.

Proof.

- we can enumerate all the functions $\mathbb{N} \to \mathbb{N}$ programmable in Agda: f_n,

Theorem
Agda is a programming language in which every programmable function is total, therefore there is a total computable function which cannot be implemented.

Proof.

- we can enumerate all the functions $\mathbb{N} \to \mathbb{N}$ programmable in Agda: f_n,
- the function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $g(n, k) = f_n(k)$ is computable,

Termination

Theorem
Agda is a programming language in which every programmable function is total, therefore there is a total computable function which cannot be implemented.

Proof.

- we can enumerate all the functions $\mathbb{N} \to \mathbb{N}$ programmable in Agda: f_n,
- the function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $g(n, k) = f_n(k)$ is computable,
- suppose that g can be implemented in Agda (otherwise we conclude),

Theorem
Agda is a programming language in which every programmable function is total, therefore there is a total computable function which cannot be implemented.

Proof.

- we can enumerate all the functions $\mathbb{N} \to \mathbb{N}$ programmable in Agda: f_n,
- the function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $g(n, k) = f_n(k)$ is computable,
- suppose that g can be implemented in Agda (otherwise we conclude),
- then consider the function $d : \mathbb{N} \to \mathbb{N}$ such that $d(n) = g(n, n) + 1$,

Theorem
Agda is a programming language in which every programmable function is total, therefore there is a total computable function which cannot be implemented.

Proof.

- we can enumerate all the functions $\mathbb{N} \to \mathbb{N}$ programmable in Agda: f_n,
- the function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $g(n, k) = f_n(k)$ is computable,
- suppose that g can be implemented in Agda (otherwise we conclude),
- then consider the function $d : \mathbb{N} \to \mathbb{N}$ such that $d(n) = g(n, n) + 1$,
- this function can be implemented thus $d = f_i$,

Termination

Theorem
Agda is a programming language in which every programmable function is total, therefore there is a total computable function which cannot be implemented.

Proof.

- we can enumerate all the functions $\mathbb{N} \to \mathbb{N}$ programmable in Agda: f_n,
- the function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $g(n, k) = f_n(k)$ is computable,
- suppose that g can be implemented in Agda (otherwise we conclude),
- then consider the function $d : \mathbb{N} \to \mathbb{N}$ such that $d(n) = g(n, n) + 1$,
- this function can be implemented thus $d = f_i$,
- and we have $d(i) =$. $\square$

Theorem
Agda is a programming language in which every programmable function is total, therefore there is a total computable function which cannot be implemented.

Proof.

- we can enumerate all the functions $\mathbb{N} \to \mathbb{N}$ programmable in Agda: f_n,
- the function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $g(n, k) = f_n(k)$ is computable,
- suppose that g can be implemented in Agda (otherwise we conclude),
- then consider the function $d : \mathbb{N} \to \mathbb{N}$ such that $d(n) = g(n, n) + 1$,
- this function can be implemented thus $d = f_i$,
- and we have $d(i) = g(i, i) + 1 =$. $\square$

Theorem

Agda is a programming language in which every programmable function is total, therefore there is a total computable function which cannot be implemented.

Proof.

- we can enumerate all the functions $\mathbb{N} \to \mathbb{N}$ programmable in Agda: f_n,
- the function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $g(n, k) = f_n(k)$ is computable,
- suppose that g can be implemented in Agda (otherwise we conclude),
- then consider the function $d : \mathbb{N} \to \mathbb{N}$ such that $d(n) = g(n, n) + 1$,
- this function can be implemented thus $d = f_i$,
- and we have $d(i) = g(i, i) + 1 = f_i(i) + 1 =$. $\square$

Theorem
Agda is a programming language in which every programmable function is total, therefore there is a total computable function which cannot be implemented.

Proof.

- we can enumerate all the functions $\mathbb{N} \to \mathbb{N}$ programmable in Agda: f_n,
- the function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $g(n, k) = f_n(k)$ is computable,
- suppose that g can be implemented in Agda (otherwise we conclude),
- then consider the function $d : \mathbb{N} \to \mathbb{N}$ such that $d(n) = g(n, n) + 1$,
- this function can be implemented thus $d = f_i$,
- and we have $d(i) = g(i, i) + 1 = f_i(i) + 1 = d(i) + 1$. $\square$

Termination

Nevertheless, we can reason on all computable functions (including non-terminating ones) by considering the reduction in an interpreter instead of implementing them directly in Agda.

In practice, the only thing which will fail is when trying to prove something like the correctness of Agda in Agda.

Bignum bakeoff

TODO: https://djm.cc/bignum-results.txt https://github.com/rcls/busy

Inductive types: truth

How do we define the type for **truth**?

Inductive types: truth

How do we define the type for **truth**?

```
data ⊤ : Set where
  tt : ⊤
```

Inductive types: truth

How do we define the type for **truth**?

```
data ⊤ : Set where
  tt : ⊤
```

In the standard library: `Data.Unit`.

Inductive types: truth

How do we define the type for **truth**?

```
data ⊤ : Set where
  tt : ⊤
```

In the standard library: `Data.Unit`.

We can show a theorem with it:

Inductive types: truth

How do we define the type for **truth**?

```
data ⊤ : Set where
  tt : ⊤
```

In the standard library: `Data.Unit`.

We can show a theorem with it:

```
⊤-intro : {A : Set} → A → ⊤
⊤-intro a = tt
```

Inductive types: falsity

How do we define the type for **falsity**?

Inductive types: falsity

How do we define the type for **falsity**?

```
data ⊥ : Set where
```

Inductive types: falsity

How do we define the type for **falsity**?

```
data ⊥ : Set where
```

In the standard library: `Data.Empty`.

Inductive types: falsity

How do we define the type for **falsity**?

```
data ⊥ : Set where
```

In the standard library: `Data.Empty`.

We can show a theorem with it:

Inductive types: falsity

How do we define the type for **falsity**?

```
data ⊥ : Set where
```

In the standard library: `Data.Empty`.

We can show a theorem with it:

```
⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()
```

Lists

We can define polymorphic types such as **lists**:

Lists

We can define polymorphic types such as **lists**:

```
data List (A : Set) : Set where
  nil  : List A
  cons : A → List A → List A
```

Lists

We can define polymorphic types such as **lists**:

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A → List A → List A
```

Lists

We can define polymorphic types such as **lists**:

```
data List (A : Set) : Set where
  []  : List A
  _∷_ : A → List A → List A
```

In the standard library: `Data.List`.

Lists

Usual functions such as **concatenation** can then be defined inductively:

Lists

Usual functions such as **concatenation** can then be defined inductively:

```
concat : {A : Set} → List A → List A → List A
concat []       m = m
concat (x ∷ l) m = x ∷ (concat l m)
```

We can program a function which computes the **tail** of a list:

Lists

We can program a function which computes the **tail** of a list:

```
tail : {A : Set} → List A → List A
tail []       = []
tail (x ∷ l) = l
```

Lists

We can program a function which computes the **tail** of a list:

```
tail : {A : Set} → List A → List A
tail []       = []
tail (x ∷ l) = l
```

but for the **head** we have a real problem:

We can program a function which computes the **tail** of a list:

```
tail : {A : Set} → List A → List A
tail []        = []
tail (x ∷ l) = l
```

but for the **head** we have a real problem:

```
head : {A : Set} → List A → A
head []        = ???
head (x ∷ l) = x
```

Lists

We could use the **Maybe** type (= `'a option` in OCaml):

Lists

We could use the **Maybe** type (= `'a option` in OCaml):

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A
```

We could use the **Maybe** type (= `'a option` in OCaml):

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A
```

(`Data.Maybe` in the standard library) and program

We could use the **Maybe** type (= `'a option` in OCaml):

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A
```

(`Data.Maybe` in the standard library) and program

```
head : {A : Set} → List A → Maybe A
head []       = nothing
head (x ∷ l) = just x
```

But this is not practical: we have to match to see whether we have `just` or a `nothing` each time we use it. It would be much better to restrict the function to non-empty lists!

Part V

Dependent types

Dependent types

In Agda, we have **polymorphic types** where a type depends on another type:

```
List A
```

Dependent types

In Agda, we have **polymorphic types** where a type depends on another type:

```
List A
```

We also have **dependent types** where a type depends on a term:

```
Vec A n
```

Vectors

We can define the type of **vectors** which are lists of a given length:

Vectors

We can define the type of **vectors** which are lists of a given length:

```
data Vec (A : Set) : (n : ℕ) → Set where
  []  : Vec A zero
  _∷_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Vectors

We can define the type of **vectors** which are lists of a given length:

```
data Vec (A : Set) : (n : ℕ) → Set where
  []  : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

In the type `Vec`, we have both

- a **parameter**: A
- an **index**: n

Indices are roughly the same as parameters, excepting they can vary between constructors.

Dependent functions

In a **dependent function**, the returned type depends on the input:

```
(x : A) → B
```

where `x` is allowed to occur in `B`.

Dependent functions

In a **dependent function**, the returned type depends on the input:

```
(x : A) → B
```

where `x` is allowed to occur in `B`.

From a logical point of view, this can be read as

$$\forall x \in A.B$$

Dependent functions

In a **dependent function**, the returned type depends on the input:

```
(x : A) → B
```

where `x` is allowed to occur in `B`.

From a logical point of view, this can be read as

$$\forall x \in A . B$$

In particular, when `x` does not occur in `B`, we can simply write

```
A → B
```

Dependent functions

In a **dependent function**, the returned type depends on the input:

`(x : A) → B`

where `x` is allowed to occur in `B`.

From a logical point of view, this can be read as

$$\forall x \in A . B$$

In particular, when `x` does not occur in `B`, we can simply write

`A → B`

NB: for multiple abstractions, we can write

`(x y : A) → B` instead of `(x : A) → (y : A) → B`

Dependent functions

For instance, we can program a function which returns a vector of `n` zeros:

Dependent functions

For instance, we can program a function which returns a vector of `n` zeros:

```
zeros : (n : ℕ) → Vec ℕ n
zeros zero    = []
zeros (suc n) = 0 ∷ (zeros n)
```

Dependent functions

For instance, we can program a function which returns a vector of `n` zeros:

```
zeros : (n : ℕ) → Vec ℕ n
zeros zero    = []
zeros (suc n) = 0 ∷ (zeros n)
```

Note that typing ensures that the resulting list is of the right length!

```
zeros : (n : ℕ) → Vec ℕ n
zeros zero    = []
zeros (suc n) = zeros n
```

raises the following error:

```
n != suc n of type ℕ
when checking that the expression zeros n has type Vec ℕ (suc n)
```

Dependent pattern matching

Agda implements an algorithm of **dependent pattern matching**: it automatically removes the cases which are not possible because of typing.

For instance, let's program the head function on vectors:

Dependent pattern matching

Agda implements an algorithm of **dependent pattern matching**: it automatically removes the cases which are not possible because of typing.

For instance, let's program the head function on vectors:

```
head : {A : Set} {n : ℕ} → Vec A (suc n) → A
head (x ∷ l) = x
```

There is no case for `[]` in the pattern matching!

Typechecking and reduction

We can also program concatenation:

Typechecking and reduction

We can also program concatenation:

```
concat : {A : Set} → {m n : ℕ} → Vec A m → Vec A n → Vec A (m + n)
concat []       l' = l'
concat (x ∷ l) l' = x ∷ (concat l l')
```

Typechecking and reduction

We can also program concatenation:

```
concat : {A : Set} → {m n : ℕ} → Vec A m → Vec A n → Vec A (m + n)
concat []       l' = l'
concat (x ∷ l) l' = x ∷ (concat l l')
```

Note that in the first case, we provide a `Vec A n` instead of a `Vec A (0 + n)`: the terms are considered modulo reduction!

Typechecking and reduction

We can also program concatenation:

```
concat : {A : Set} → {m n : ℕ} → Vec A m → Vec A n → Vec A (m + n)
concat []       l' = l'
concat (x ∷ l) l' = x ∷ (concat l l')
```

Note that in the first case, we provide a `Vec A n` instead of a `Vec A (0 + n)`: the terms are considered modulo reduction!

This is also visible on the following test:

```
l : Vec ℕ (3 + 1)
l = concat (0 ∷ (0 ∷ [])) (0 ∷ (0 ∷ []))
```

(a vector of length `2 + 2` is a vector of length `3 + 1`).

Finite sets

The use of vectors has solved the problem for `head`, but suppose that we want to define the function `ith`:

```
ith : {A : Set} → (i : ℕ) → {n : ℕ} → (l : Vec A n) → A
ith i l = ?
```

What is the problem?

Finite sets

The use of vectors has solved the problem for `head`, but suppose that we want to define the function `ith`:

```
ith : {A : Set} → (i : ℕ) → {n : ℕ} → (l : Vec A n) → A
ith i l = ?
```

What is the problem?

We can generalize the trick we used for head:

```
ith : {A : Set} {n : ℕ} → (i : ℕ) → Vec A (suc (i + n)) → A
ith zero    (x ∷ l) = x
ith (suc i) (x ∷ l) = ith i l
```

Finite sets

The use of vectors has solved the problem for `head`, but suppose that we want to define the function `ith`:

```
ith : {A : Set} → (i : ℕ) → {n : ℕ} → (l : Vec A n) → A
ith i l = ?
```

What is the problem?

We could add an extra condition, but this is a bit heavy on the long run:

```
ith : {A : Set} → (i : ℕ) → {n : ℕ} → (l : Vec A n) → (p : i < n) → A
ith zero    (x ∷ l) p = x
ith (suc i) (x ∷ l) p = ith i l (≤-pred p)
```

Finite set

Consider the sets

$$F_n = \{0, \ldots, n-1\}$$

an inductive definition is given by

$$F_0 = \emptyset$$

and

Finite set

Consider the sets

$$F_n = \{0, \ldots, n-1\}$$

an inductive definition is given by

$$F_0 = \emptyset$$

and

$$F_{n+1} = F_n \sqcup \{n\}$$

Finite set

Consider the sets

$$F_n = \{0, \ldots, n-1\}$$

an inductive definition is given by

$$F_0 = \emptyset$$

and

$$F_{n+1} = \{0\} \sqcup \{1 + i \mid i \in F_n\}$$

Finite set

Consider the sets

$$F_n = \{0, \dots, n-1\}$$

an inductive definition is given by

$$F_0 = \emptyset$$

and

$$F_{n+1} = \{0\} \sqcup F_n$$

Finite set

Consider the sets

$$F_n = \{0, \dots, n-1\}$$

an inductive definition is given by

$$F_0 = \emptyset$$

and

$$F_{n+1} = \{0\} \sqcup F_n$$

This means that

- 0 belongs to any set F_{n+1},
- any element of F_n induces an element of F_{n+1}.

Finite sets

It is thus natural, elegant and practical to define the type

of natural numbers between `0` (inclusive) and n (exclusive), see `Data.Fin`.

Finite sets

It is thus natural, elegant and practical to define the type

```
data Fin : ℕ → Set where
  Fin-zero : {n : ℕ} → Fin (suc n)
  Fin-suc  : {n : ℕ} → Fin n → Fin (suc n)
```

of natural numbers between `0` (inclusive) and n (exclusive), see `Data.Fin`.

Finite sets

It is thus natural, elegant and practical to define the type

```
data Fin : ℕ → Set where
  zero : Fin (suc zero)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

of natural numbers between `0` (inclusive) and n (exclusive), see `Data.Fin`.

Finite sets

It is thus natural, elegant and practical to define the type

```
data Fin : ℕ → Set where
  zero : Fin (suc zero)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

of natural numbers between `0` (inclusive) and n (exclusive), see `Data.Fin`.

We can then define

```
ith : {A : Set} → {n : ℕ} → Fin n → Vec A n → A
ith zero     (x ∷ l) = x
ith (suc i) (x ∷ l) = ith i l
```

Part VI

Logic

So far, we have seen that Agda is a very expressive programming language.

By Curry-Howard, we can also see it as a proof assistant.

In order to do real logic, we need some more **connectives** and in particular

$\mathfrak{equality}$.

Implication

Recall that **implication**

$$A \Rightarrow B$$

is

Implication

Recall that **implication**

$$A \Rightarrow B$$

is (non-dependent) arrow type

```
A → B
```

Truth and falsity

Recall that the types for **truth** and **falsity** are respectively

Truth and falsity

Recall that the types for **truth** and **falsity** are respectively

```
data ⊤ : Set where
  tt : ⊤
```

and

Truth and falsity

Recall that the types for **truth** and **falsity** are respectively

```
data ⊤ : Set where
  tt : ⊤
```

and

```
data ⊥ : Set where
```

In `Data.Unit` and `Data.Empty`.

Negation

As usual, **negation** can be defined as

Negation

As usual, **negation** can be defined as

```
¬ : Set → Set
¬ A = A → ⊥
```

In `Relation.Nullary`.

Note how wonderful it is to have a type `Set`.

Conjunction

Conjunction is given by

Conjunction

Conjunction is given by **product**:

```
data prod (A B : Set) : Set where
  pair : A → B → prod A B
```

Conjunction

Conjunction is given by **product**:

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

Conjunction

Conjunction is given by **product**:

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

and we have already seen this in our first example:

```
×-comm : {A B : Set} → (A × B) → (B × A)
×-comm (a , b) = (b , a)
```

Conjunction

Conjunction is given by **product**:

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

and we have already seen this in our first example:

```
×-comm : {A B : Set} → (A × B) → (B × A)
×-comm (a , b) = (b , a)
```

Projections are defined by pattern-matching:

Conjunction

Conjunction is given by **product**:

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

and we have already seen this in our first example:

```
×-comm : {A B : Set} → (A × B) → (B × A)
×-comm (a , b) = (b , a)
```

Projections are defined by pattern-matching:

```
fst : {A B : Set} → A × B → A
fst (a , b) = a
```

which is a proof of $(A \wedge B) \Rightarrow A$.

Disjunction

Disjunction is given by

Disjunction

Disjunction is given by **coproduct**:

```
data _⊎_ (A B : Set) : Set where
  left  : A → A ⊎ B
  right : B → A ⊎ B
```

Disjunction

Disjunction is given by **coproduct**:

```
data _⊎_ (A B : Set) : Set where
  left  : A → A ⊎ B
  right : B → A ⊎ B
```

It is also commutative:

Disjunction

Disjunction is given by **coproduct**:

```
data _⊎_ (A B : Set) : Set where
  left  : A → A ⊎ B
  right : B → A ⊎ B
```

It is also commutative:

```
⊎-comm : {A B : Set} → A ⊎ B → B ⊎ A
⊎-comm (left a)  = right a
⊎-comm (right b) = left b
```

Decidable types

Recall that we are in intuitionistic logic: `A ⊎ ¬ A` does not hold for every type `A`.

Decidable types

Recall that we are in intuitionistic logic: `A ⊎ ¬ A` does not hold for every type `A`.

A type for which this holds is called **decidable**:

```
Dec : Set → Set
Dec A = A ⊎ ¬ A
```

We will see an example later on.

Predicates

Usually, a predicate P on a set A is encoded as

Predicates

Usually, a predicate P on a set A is encoded as a function

$$A \to \{0, 1\}$$

Predicates

Usually, a predicate P on a set A is encoded as a function

$$A \to \{0, 1\}$$

In Agda, we could thus encode a predicate on a type `A` as

Predicates

Usually, a predicate P on a set A is encoded as a function

$$A \to \{0, 1\}$$

In Agda, we could thus encode a predicate on a type `A` as a function

```
A → bool
```

Predicates

Usually, a predicate P on a set A is encoded as a function

$$A \to \{0, 1\}$$

In Agda, we could thus encode a predicate on a type `A` as a function

```
A → bool
```

This is however not satisfactory, why?

Predicates

The standard way of encoding a **predicate** `P` on a type `A` is as an element of type

```
A → Set
```

Given a term `a` of type `A`, the type

```
P a
```

is the type of all proofs such that `P a` holds.

Predicates

We can define predicates inductively and reason about them by pattern matching!

For instance, let's define a predicate on natural numbers corresponding to "being even":

Predicates

We can define predicates inductively and reason about them by pattern matching!

For instance, let's define a predicate on natural numbers corresponding to "being even":

```
data Even : ℕ → Set where
  even-zero : Even zero
  even-suc  : {n : ℕ} → Even n → Even (suc (suc n))
```

Predicates

We can define predicates inductively and reason about them by pattern matching!

For instance, let's define a predicate on natural numbers corresponding to "being even":

```
data Even : ℕ → Set where
  even-zero : Even zero
  even-suc  : {n : ℕ} → Even n → Even (suc (suc n))
```

We can then show that 4 is even:

```
four-even : Even (suc (suc (suc (suc zero))))
four-even = even-suc (even-suc even-zero)
```

Predicates

We can define predicates inductively and reason about them by pattern matching!

For instance, let's define a predicate on natural numbers corresponding to "being even":

```
data Even : ℕ → Set where
  even-zero : Even zero
  even-suc  : {n : ℕ} → Even n → Even (suc (suc n))
```

We can then show that 1 is not even:

```
one-not-even : Even (suc zero) → ⊥
one-not-even ()
```

Predicates

We can define predicates inductively and reason about them by pattern matching!

For instance, let's define a predicate on natural numbers corresponding to "being even":

```
data Even : ℕ → Set where
  even-zero : Even zero
  even-suc  : {n : ℕ} → Even n → Even (suc (suc n))
```

We can then show that 3 is not even:

```
three-not-even : Even (suc (suc (suc zero))) → ⊥
three-not-even (even-suc ())
```

Predicates

We can similarly define predicates with other arities.

For instance, the order on natural numbers is

Predicates

We can similarly define predicates with other arities.

For instance, the order on natural numbers is

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : {n : ℕ} → zero ≤ n
  s≤s : {m n : ℕ} → m ≤ n → suc m ≤ suc n
```

Predicates

We can similarly define predicates with other arities.

For instance, the order on natural numbers is

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : {n : ℕ} → zero ≤ n
  s≤s : {m n : ℕ} → m ≤ n → suc m ≤ suc n
```

and we can show that it is transitive by

```
≤-trans : {m n o : ℕ} → m ≤ n → n ≤ o → m ≤ o
≤-trans z≤s     l       = z≤s
≤-trans (s≤s k) (s≤s l) = s≤s (≤-trans k l)
```

Predicates

We can similarly define predicates with other arities.

For instance, the order on natural numbers is

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : {n : ℕ} → zero ≤ n
  s≤s : {m n : ℕ} → m ≤ n → suc m ≤ suc n
```

Note that it could also be defined as a function, but this is less natural:

```
_≤_ : ℕ → ℕ → Set
zero  ≤ n     = ⊤
suc m ≤ zero  = ⊥
suc m ≤ suc n = m ≤ n
```

Equality

Magically, we can even define **propositional equality**:

Equality

Magically, we can even define **propositional equality**:

```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x
```

Equality

Magically, we can even define **propositional equality**:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Equality

Magically, we can even define **propositional equality**:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

What's going to happen when we reason by induction on equality?

Equality

Magically, we can even define **propositional equality**:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

What's going to happen when we reason by induction on equality?

Note that there are two notions of equality in Agda:

- *definitional equality*: terms are considered up to β-reduction (e.g. `2+2` $=$ `3+1`),
- *propositional equality*: the one above.

Equality

Magically, we can even define **propositional equality**:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

What's going to happen when we reason by induction on equality?

Note that there are two notions of equality in Agda:

- *definitional equality*: terms are considered up to β-reduction (e.g. `2+2` = `3+1`),
- *propositional equality*: the one above.

It is defined in `Relation.Binary.PropositionalEquality`.

Basic properties of equality

Let's show some basic properties of equality. It is

- **reflexive**: this is what the `refl` constructor is saying,

Basic properties of equality

Let's show some basic properties of equality. It is

- **reflexive**: this is what the `refl` constructor is saying,
- **symmetric**:

Basic properties of equality

Let's show some basic properties of equality. It is

- **reflexive**: this is what the `refl` constructor is saying,
- **symmetric**:

```
sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl
```

Basic properties of equality

Let's show some basic properties of equality. It is

- **reflexive**: this is what the `refl` constructor is saying,
- **symmetric**:

  ```
  sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
  sym refl = refl
  ```

- **transitive**:

Basic properties of equality

Let's show some basic properties of equality. It is

- **reflexive**: this is what the `refl` constructor is saying,
- **symmetric**:

```
sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl
```

- **transitive**:

```
trans : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

Basic properties of equality

Equality is a **congruence**:

Basic properties of equality

Equality is a **congruence**:

```
cong : {A B : Set} {x y : A} (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

Basic properties of equality

Equality is a **congruence**:

```
cong : {A B : Set} {x y : A} (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

For instance,

```
cong12 : {m n : ℕ} → m ≡ n → (m + 12) ≡ (n + 12)
cong12 p = cong (λ k → k + 12) p
```

Basic properties of equality

We can use this to show that addition is associative:

Basic properties of equality

We can use this to show that addition is associative:

```
+-assoc : (m n p : ℕ) → ((m + n) + p) ≡ (m + (n + p))
+-assoc zero    n p = refl
+-assoc (suc m) n p = cong suc (+-assoc m n p)
```

Substitutivity

A last important property is that equality is **substitutive**:

Substitutivity

A last important property is that equality is **substitutive**:

```
subst : {A : Set} (P : A → Set) → {x y : A} → x ≡ y → P x → P y
subst P refl p = p
```

If two things are equal and one satisfies a property then the other also does.

This is also sometimes called **transport**.

Coercion

In particular, we can **coerce** a term of a given type into one of some equal type:

Coercion

In particular, we can **coerce** a term of a given type into one of some equal type:

```
coe : {A B : Set} → A ≡ B → A → B
coe refl x = x
```

In particular, we can **coerce** a term of a given type into one of some equal type:

```
coe : {A B : Set} → A ≡ B → A → B
coe e x = subst (λ A → A) e x
```

Decidability

Recall the definition of A being **decidable**:

```
Dec A = A ⊎ ¬ A
```

Decidability

Recall the definition of A being **decidable**:

```
Dec A = A ⊎ ¬ A
```

For instance, on booleans, equality is decidable:

Decidability

Recall the definition of *A* being **decidable**:

```
Dec A = A ⊎ ¬ A
```

For instance, on booleans, equality is decidable:

```
Bool-≡-dec : (a b : Bool) → Dec (a ≡ b)
Bool-≡-dec false false = left refl
Bool-≡-dec false true  = right (λ ())
Bool-≡-dec true  false = right (λ ())
Bool-≡-dec true  true  = left refl
```

Decidability

Recall the definition of A being **decidable**:

```
Dec A = A ⊎ ¬ A
```

For instance, on natural numbers, equality is decidable:

Decidability

Recall the definition of *A* being **decidable**:

```
Dec A = A ⊎ ¬ A
```

For instance, on natural numbers, equality is decidable:

```
ℕ-≡-dec : (m n : ℕ) → Dec (m ≡ n)
ℕ-≡-dec zero    zero    = left refl
ℕ-≡-dec zero    (suc n) = right (λ ())
ℕ-≡-dec (suc m) zero    = right (λ ())
ℕ-≡-dec (suc m) (suc n) with ℕ-≡-dec m n
ℕ-≡-dec (suc m) (suc n) | left e   = left  (cong suc e)
ℕ-≡-dec (suc m) (suc n) | right e' = right (λ e → e' (suc-injective e))
```

Auxiliary matching

We have learned a new syntax to match on auxiliary computations:

```
f : A → B
f x with e
... | v = result
```

Decidability

Equality is *not* always decidable.

For instance,

Decidability

Equality is *not* always decidable.

For instance, how would we decide the equality of two functions $\mathbb{N} \to \mathbb{N}$?

Decidability

Equality is *not* always decidable.

For instance, how would we decide the equality of two functions $\mathbb{N} \to \mathbb{N}$?

By the way, equality on functions is not **extensional**, we only have the implication

Decidability

Equality is *not* always decidable.

For instance, how would we decide the equality of two functions ℕ → ℕ?

By the way, equality on functions is not **extensional**, we only have the implication

```
extfun : {A B : Set} → {f g : A → B} → f ≡ g → (a : A) → f a ≡ g a
extfun refl a = refl
```

Decidability

Equality is *not* always decidable.

For instance, how would we decide the equality of two functions `ℕ → ℕ`?

By the way, equality on functions is not **extensional**, we only have the implication

```
extfun : {A B : Set} → {f g : A → B} → f ≡ g → (a : A) → f a ≡ g a
extfun refl a = refl
```

but not the converse

```
funext : {A B : Set} → {f g : A → B} → ((a : A) → f a ≡ g a) → f ≡ g
```

Part VII

Extrinsic vs intrinsic proofs

Extrinsic vs intrinsic proofs

There are two approaches when proving that a program is correct.

- **Extrinsic** approach:

Extrinsic vs intrinsic proofs

There are two approaches when proving that a program is correct.

- **Extrinsic** approach:
    1. we program the function we are interested in,

Extrinsic vs intrinsic proofs

There are two approaches when proving that a program is correct.

- **Extrinsic** approach:
  1. we program the function we are interested in,
  2. we show that it is correct.

Extrinsic vs intrinsic proofs

There are two approaches when proving that a program is correct.

- **Extrinsic** approach:
  1. we program the function we are interested in,
  2. we show that it is correct.
- **Intrinsic** approach: we directly define the function we are interested in with a type which guarantees its correctness.

Length of concatenation: extrinsic approach

Suppose that we want to show that concatenation adds the length of lists.

Length of concatenation: extrinsic approach

Suppose that we want to show that concatenation adds the length of lists.

We define the length function on lists:

Length of concatenation: extrinsic approach

Suppose that we want to show that concatenation adds the length of lists.

We define the length function on lists:

```
length : {A : Set} → List A → ℕ
length []       = 0
length (x ∷ l) = 1 + (length l)
```

Length of concatenation: extrinsic approach

Suppose that we want to show that concatenation adds the length of lists.

We define the length function on lists:

```
length : {A : Set} → List A → ℕ
length []       = 0
length (x ∷ l) = 1 + (length l)
```

and show that the property is satisfied:

Length of concatenation: extrinsic approach

Suppose that we want to show that concatenation adds the length of lists.

We define the length function on lists:

```
length : {A : Set} → List A → ℕ
length []       = 0
length (x ∷ l) = 1 + (length l)
```

and show that the property is satisfied:

```
concat-length : {A : Set} → (k l : List A) →
                length (concat k l) ≡ length k + length l
concat-length []      l = refl
concat-length (x ∷ k) l = cong suc (concat-length k l)
```

Length of concatenation: intrinsic approach

In the intrinsic approach, we define an adapted type (vectors = lists of given length) and meaningfully type concatenation:

Length of concatenation: intrinsic approach

In the intrinsic approach, we define an adapted type (vectors = lists of given length) and meaningfully type concatenation:

```
concat : {A : Set} → {m n : ℕ} → Vec A m → Vec A n → Vec A (m + n)
concat []       l' = l'
concat (x ∷ l) l' = x ∷ (concat l l')
```

Part VIII

Dependent sums

Dependent sums

For vectors, we were easily able to inductively define a type of lists of a given length.

This case is however particular: given

- a type `A : Set`,
- a predicate `P : A → Set`,

we would like to define the set of elements `a` of type `A` such that `P a` holds.

In set-theoretic notation: $\{a \in A \mid P\ a\}$.

Dependent sums

For vectors, we were easily able to inductively define a type of lists of a given length.

This case is however particular: given

- a type `A : Set`,
- a predicate `P : A → Set`,

we would like to define the set of elements `a` of type `A` such that `P a` holds.

In set-theoretic notation: $\{a \in A \mid P\ a\}$.

Because we are constructive, we want to implement it as pairs consisting of

- an element `a` of type `A`,
- an element `p` of type `P a`.

Looks like a product!

Dependent sums

The type for **products** is

Dependent sums

The type for **products** is

```
data prod (A B : Set) : Set where
  pair : A → B → prod A B
```

Dependent sums

The type for **products** is

```
data prod (A B : Set) : Set where
  pair : A → B → prod A B
```

The type for **dependent sums** is

Dependent sums

The type for **products** is

```
data prod (A B : Set) : Set where
  pair : A → B → prod A B
```

The type for **dependent sums** is

```
data dsum (A : Set) (P : A → Set) : Set where
  pair : (a : A) → P a → dsum A P
```

Dependent sums

The type for **products** is

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

The type for **dependent sums** is

```
data Σ (A : Set) (P : A → Set) : Set where
  _,_ : (a : A) → P a → Σ A P
```

Dependent sums

The type for **products** is

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

The type for **dependent sums** is

```
data Σ (A : Set) (P : A → Set) : Set where
  _,_ : (a : A) → P a → Σ A P
```

Defined in `Data.Product`.

Dependent sums

The **dependent sums** can be noted

- in Agda: `Σ A P`,

Dependent sums

The **dependent sums** can be noted

- in Agda: `Σ A P`,
- in maths: $\sum_{\mathtt{a} \in \mathtt{A}} \mathtt{P(a)}$,

Dependent sums

The **dependent sums** can be noted

- in Agda: `Σ A P`,
- in maths: $\sum_{\mathtt{a} \in \mathtt{A}} \mathtt{P(a)}$,
- in set theory: $\{\mathtt{a} \in \mathtt{A} \mid \mathtt{P\ a}\}$,

Dependent sums

The **dependent sums** can be noted

- in Agda: `Σ A P`,
- in maths: $\sum_{\mathtt{a} \in \mathtt{A}} \mathtt{P(a)}$,
- in set theory: $\{\mathtt{a} \in \mathtt{A} \mid \mathtt{P\ a}\}$,
- in logic: $\exists \mathtt{a} \in \mathtt{A}.(\mathtt{P\ a})$.

Dependent sums

The **dependent sums** can be noted

- in Agda: $\texttt{Σ A P}$,
- in maths: $\sum_{\mathtt{a} \in \mathtt{A}} \mathtt{P(a)}$,
- in set theory: $\{\mathtt{a} \in \mathtt{A} \mid \mathtt{P\ a}\}$,
- in logic: $\exists \mathtt{a} \in \mathtt{A}.(\mathtt{P\ a})$.

(this is a constructive variant of those)

Dependent sums

The **dependent sums** can be noted

- in Agda: `Σ A P`,
- in maths: $\sum_{\mathtt{a} \in \mathtt{A}} \mathtt{P(a)}$,
- in set theory: $\{\mathtt{a} \in \mathtt{A} \mid \mathtt{P\ a}\}$,
- in logic: $\exists \mathtt{a} \in \mathtt{A}.(\mathtt{P\ a})$.

(this is a constructive variant of those)

Note that products are a particular case of dependent sum:

```
A × B    =    Σ A (λ _ → B )
```

like in maths, we have

$$m \times n = \sum_{i \in \{1,\ldots,m\}} n_i$$

Vectors

We could have defined vectors as

```
Vec' : Set → ℕ → Set
Vec' A n = Σ (List A) (λ l → length l ≡ n)
```

In theory, this is as powerful as the above type.

In practice, the less equalities you have the better you are.

Vectors

We could have defined finite sets as

```
Fin' : ℕ → Set
Fin' n = Σ ℕ (λ i → i < n)
```

In theory, this is as powerful as the above type.

In practice, the less inequalities you have the better you are.

Division

The usual way of implementing Euclidean division in OCaml:

Division

The usual way of implementing Euclidean division 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)
```

Division

The usual way of implementing Euclidean division 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)
```

The type for division in the intrinsic approach in Agda would be:

Division

The usual way of implementing Euclidean division 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)
```

The type for division in the intrinsic approach in Agda would be:

```
div : (m n : ℕ) → Σ ℕ (λ q → Σ ℕ (λ r → (m ≡ n * q + r) × (r < n)))
div m n = ?
```

Division

The usual way of implementing Euclidean division 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)
```

The type for division in the intrinsic approach in Agda would be:

```
div : (m n : ℕ) → Σ ℕ (λ q → Σ ℕ (λ r → (m ≡ n * q + r) × (r < n)))
div m n = ?
```

Why can't we directly translate the above code?

Half of even numbers

Remember that we have defined the predicate of being even inductively as

```
data Even : ℕ → Set where
  even-zero : Even zero
  even-suc  : {n : ℕ} → Even n → Even (suc (suc n))
```

Half of even numbers

Remember that we have defined the predicate of being even inductively as

```
data Even : ℕ → Set where
  even-zero : Even zero
  even-suc  : {n : ℕ} → Even n → Even (suc (suc n))
```

We can then show that every even number has a half by induction by

Half of even numbers

Remember that we have defined the predicate of being even inductively as

```
data Even : ℕ → Set where
  even-zero : Even zero
  even-suc  : {n : ℕ} → Even n → Even (suc (suc n))
```

We can then show that every even number has a half by induction by

```
even-half : {n : ℕ} → Even n → Σ ℕ (λ m → m + m ≡ n)
even-half even-zero = zero , refl
even-half (even-suc e) with even-half e
even-half (even-suc _) | m , e = (suc m) , cong suc (trans (+-suc m m) (cong
```

Half of even numbers

Remember that we have defined the predicate of being even inductively as

```
data Even : ℕ → Set where
  even-zero : Even zero
  even-suc  : {n : ℕ} → Even n → Even (suc (suc n))
```

We can then show that every even number has a half by induction by

```
even-half : {n : ℕ} → Even n → Σ ℕ (λ m → m + m ≡ n)
even-half even-zero = zero , refl
even-half (even-suc e) with even-half e
even-half (even-suc _) | m , e = (suc m) , cong suc (trans (+-suc m m) (cong
```

We can finally compute the half of four by normalizing

Half of even numbers

Remember that we have defined the predicate of being even inductively as

```
data Even : ℕ → Set where
  even-zero : Even zero
  even-suc  : {n : ℕ} → Even n → Even (suc (suc n))
```

We can then show that every even number has a half by induction by

```
even-half : {n : ℕ} → Even n → Σ ℕ (λ m → m + m ≡ n)
even-half even-zero = zero , refl
even-half (even-suc e) with even-half e
even-half (even-suc _) | m , e = (suc m) , cong suc (trans (+-suc m m) (cong
```

We can finally compute the half of four by normalizing

```
two : Σ ℕ _
two = even-half four-even
```

Dependent product types

The Agda type

```
(a : A) → B
```

is called a **dependent product** type and noted

$$\prod_{a \in A} B(a)$$

Dependent sum and product types

The dependent sum and product types are satisfy dual properties:

	Sum	Product
Agda	`Σ A (λ a → B)`	`(a : A) → B`
Maths	$\sum_{a \in A} B(a)$	$\prod_{a \in A} B(a)$
Logic	$\exists a \in A.B(a)$	$\forall a \in A.B(a)$
Non-dependent	$A \times B$	$A \Rightarrow B$

Part IX

Records

Records

Records are tuples with named fields.

For instance, we can model a person by

```
(** A person: first name, last name, age, height *)
type person = string * string * float * float
```

This is not very practical because:

- we can confuse between fields
- it is not easy to extract fields

  ```
  let (first, last, age, height) = person in ...
  ```
- the resulting code is not very robust (reordering fields, adding new fields, etc.)

Records

Records are tuples with named fields.

We thus define

```
type person =
  {
    first  : string;
    last   : string;
    age    : float;
    height : float
  }
```

and we can access to the age by

```
let a = person.age in ...
```

Pairs

The usual definition of the product of types is

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

Pairs

The usual definition of the product of types is

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

But we could alternatively define it as a record:

Pairs

The usual definition of the product of types is

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

But we could alternatively define it as a record:

```
record _×_ (A : Set) (B : Set) : Set where
  field
    fst : A
    snd : B
```

Pairs

The usual definition of the product of types is

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

But we could alternatively define it as a record:

```
record _×_ (A : Set) (B : Set) : Set where
  field
    fst : A
    snd : B
```

and use it as expected:

```
proj₁ : {A B : Set} → prod A B → A
proj₁ p = prod.fst p
```

Pairs

If we define products as

```
record _×_ (A : Set) (B : Set) : Set where

  field
    fst : A
    snd : B
```

constructing values is heavy:

```
pair : {A B : Set} → A → B → prod A B
pair a b = record { fst = a ; snd = b }
```

Pairs

If we define products as

```
record _×_ (A : Set) (B : Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B
```

constructing values is easy:

```
pair : {A B : Set} → A → B → prod A B
pair a b = a , b
```

Records

As usual in Agda, everything is dependent.

This means that the type of a field can depend on a previous field!

Groups

For instance, we can define groups as

```
record Group : Set₁ where
  field
    X : Set
    _·_ : X → X → X
    e : X
    i : X → X
    assoc  : (x y z : X) → (x · y) · z ≡ x · (y · z)
    unit-l : (x : X) → e · x ≡ x
    unit-r : (x : X) → x · e ≡ x
    inv-l  : (x : X) → i x · x ≡ e
    inv-r  : (x : X) → x · i x ≡ e
```

Groups

We can then show classical math results:

```
inv-u-l : {G : Group} → (x x' y : X) → x · y ≡ e → x' · y ≡ e → x ≡ x'
inv-u-l x x' y p q = trans (sym (unit-r x)) ?
```

but this quickly gets difficult to read.

Fortunately, Agda has a syntax for you!

Groups

We can then show classical math results:

```
inv-u-l : {G : Group} → (x x' y : X) → x · y ≡ e → x' · y ≡ e → x ≡ x'
inv-u-l x x' y p q = begin
  x               ≡⟨ sym (unit-r x) ⟩
  x · e           ≡⟨ cong (λ y → x · y) (sym (inv-r y)) ⟩
  x · (y · i y)   ≡⟨ sym (assoc x y (i y)) ⟩
  (x · y) · i y   ≡⟨ cong (λ x → x · i y) p ⟩
  e · i y         ≡⟨ cong (λ x → x · i y) (sym q) ⟩
  (x' · y) · i y  ≡⟨ assoc x' y (i y) ⟩
  x' · (y · i y)  ≡⟨ cong (λ y → x' · y) (inv-r y) ⟩
  x' · e          ≡⟨ unit-r x' ⟩
  x'              ∎
```