CSC_51051_EP: Computational logic from Artificial intelligence to Zero bugs

Samuel Mimram

2025

École polytechnique

Let's start with some polls

- English or French?

Let's start with some polls

- English or French?
- Who has followed INF412?

Let's start with some polls

- English or French?
- Who has followed INF412?
- Who has already used OCaml?

Let's start with some polls

- English or French?
- Who has followed INF412?
- Who has already used OCaml?
- Who has already heard of a proof assistant?

What is this course about?

PROGRAM

=

PROOF

What is this course about?

A rough history of the subject.

- 1900: formalization of the notion of proof
  Hilbert, Frege, Russell, Brouwer, Gentzen, ...
- 1930: functional programming (λ-calculus)
  Church, ...
- 1960: typing rules for functional programming = rules for logic
  Curry, Howard, ...
- 1970: programs to verify proofs
  de Bruijn, Coquand, ...
- 1970: dependent types
  Martin-Löf, Coquand, ...

What is this course about?

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : A \to A}\,(\mathsf{ax}) \qquad \dfrac{}{\Gamma \vdash x : A \to A}\,(\mathsf{ax})}{\Gamma \vdash fx : A}\,(\to_{\mathsf{E}}) \qquad \dfrac{}{\Gamma \vdash f : A \to A}\,(\mathsf{ax})}{f : A \to A, x : A \vdash f(fx) : A}\,(\to_{\mathsf{E}})$$

$$\dfrac{f : A \to A, x : A \vdash f(fx) : A}{f : A \to A \vdash \lambda x^{A}.f(fx) : A \to A}\,(\to_{\mathsf{I}})$$

$$\dfrac{f : A \to A \vdash \lambda x^{A}.f(fx) : A \to A}{\vdash \lambda f^{A \to A}.\lambda x^{A}.f(fx) : (A \to A) \to A \to A}\,(\to_{\mathsf{I}})$$

What is this course about?

This correspondence between proofs and programs implies that

- we can automatically check whether a proof is valid or not,
- we can prove properties about programs,
- we can use programs to generate proofs.

Plan of the course

Programming proofs (labs in OCaml)

1. Typed functional programming
2. Intuitionistic propositional logic
3. λ-calculus
4. The proof-as-program correspondence

Proving programs (labs in Agda)

5. Introduction to Agda
6. First order logic
7. Dependent types I
8. Dependent types II
9. Homotopy type theory

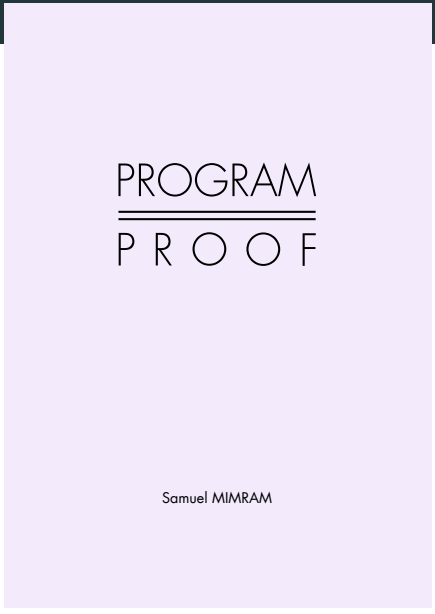
Resources

All resources can be found on the webpage of the course:

`http://inf551.mimram.fr/`

`https://moodle.polytechnique.fr/`

Course notes



Communicating

You can reach me by mail:

samuel.mimram@polytechnique.edu

Evaluation

You will be evaluated on

1. the labs (1/3)
2. the 4th lab/project (1/3)
3. an exam (1/3)

It is important that you *submit your lab solutions on moodle*, you have 1 week to do so.

(if you don't have access to it, use mail)

Evaluation

Some general remarks:

- the goal is that you understand, please **ask questions**, including for the project,
- you are strongly advised to complete (at least) the mandatory parts of labs,
- do not forget to **submit** the lab on the moodle,
- you have one week to submit or update the lab,
- copying code is considered as cheating,
- using generative AI (such as ChatGPT) to generate code is considered as cheating,
- the course focuses on theory and the lab focus on practice,
- it will be difficult for you to catch up, please attend the courses.

Part I

PROGRAM = PROOF

Testing vs proving

Programming

Most programmers use **tests** in order to validate their developments.

This is based on the belief that if the program

- uses "regular enough" functions, and
- "small" constants,

then enough tests should cover all possible behaviors.

Mathematics

No mathematician, in order to prove

$$\forall n \in \mathbb{N}. P(n)$$

will start checking $P(0)$, then $P(1)$, then $P(2)$, ...

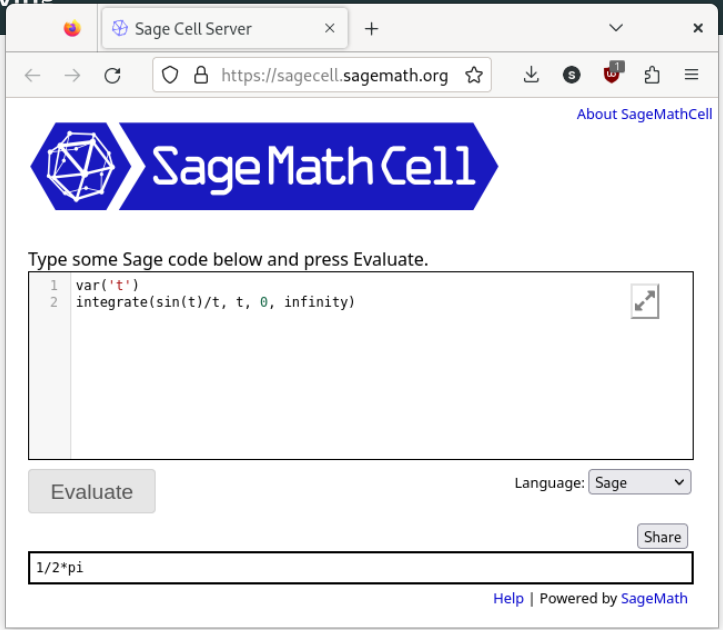
He will make a **proof**, which ensures that $P(n)$ holds whichever $n \in \mathbb{N}$ is.

Can we prove programs?

Testing vs proving

How much is $\int_0^\infty \frac{\sin(t)}{t} \mathrm{d}t$?

Testing vs proving



Testing vs proving

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

Testing vs proving

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

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

Testing vs proving

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

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

Testing vs proving

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

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

$$\int_0^\infty \frac{\sin(t)}{t} \frac{\sin(t/101)}{t/101} \frac{\sin(t/201)}{t/201} dt =$$

Testing vs proving

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

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

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

Testing vs proving

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

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

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

$$\int_0^\infty \frac{\sin(t)}{t} \frac{\sin(t/101)}{t/101} \frac{\sin(t/201)}{t/201} \frac{\sin(t/301)}{t/301} dt =$$

Testing vs proving

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

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

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

$$\int_0^\infty \frac{\sin(t)}{t} \frac{\sin(t/101)}{t/101} \frac{\sin(t/201)}{t/201} \frac{\sin(t/301)}{t/301} dt = \frac{\pi}{2}$$

Testing vs proving

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

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

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

$$\int_0^\infty \frac{\sin(t)}{t}\frac{\sin(t/101)}{t/101}\frac{\sin(t/201)}{t/201}\frac{\sin(t/301)}{t/301}\mathrm{d}t = \frac{\pi}{2}$$

$$\vdots$$

Testing vs proving

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

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

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

$$\int_0^\infty \frac{\sin(t)}{t}\frac{\sin(t/101)}{t/101}\frac{\sin(t/201)}{t/201}\frac{\sin(t/301)}{t/301}\,\mathrm{d}t = \frac{\pi}{2}$$

$$\vdots$$

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

Testing vs proving

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

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

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

$$\int_0^\infty \frac{\sin(t)}{t}\frac{\sin(t/101)}{t/101}\frac{\sin(t/201)}{t/201}\frac{\sin(t/301)}{t/301}\mathrm{d}t = \frac{\pi}{2}$$

$$\vdots$$

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

Testing vs proving

In fact, the equality

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

starts breaking at

$$n =$$

Testing vs proving

In fact, the equality

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

starts breaking at

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

Testing vs proving

Already in the 70s Dijkstra was claiming:

Program testing can be used to show the presence of bugs, but never to show their absence!

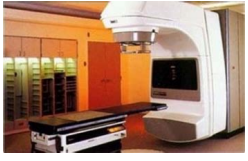
Testing vs proving

Already in the 70s Dijkstra was claiming:

Program testing can be used to show the presence of bugs, but never to show their absence!

Morale: testing rarely covers all cases, see previous example or real life.

See the usual list of software bugs:



Therac-25

Ariane 5

...

The Therac-25

The Therac-25 was

- a radiation therapy machine with two modes:
  low energy electrons (e.g. skin) and high energy X-rays (e.g. lungs),
- one day the operator pressed “x” instead of “e”, and quickly corrected to “e”
- the patient received 100 times the expected dose and eventually died
- there was a concurrency error: if an edit was performed during the magnet setting phase (8 seconds) it was not taken into account because of the value of the shared completion variable, although the screen made you think it did
- the bug was present in the Therac-20 but hardware prevented this
- there was an overflow in a 1 byte long variable which did also caused overdose under bad cirumstances every 256 attempts
- first software bug to actually kill people

Ariane 5

Ariane 5

- reused the Ariane 5 reused the inertial reference platform (SRI) from Ariane 4 (uses sensors to compute the position)
- the acceleration was 5 times bigger and converted from 64 bits to 16 bits, which resulted in an overflow
- this caused a hardware exception, which caused sending test data on the data bus
- at t0+37, the autopilot is then launched, uses the test data as actual data, and thus abruptly changes the trajectory of the rocket
- the SRI were useful only before launching and kept active during the 40 first seconds because this was required by Ariane 4.

Proof assistants

Starting from the 70s, people started to develop **proof assistants** which are programs which can check proofs (Agda, Coq, Isabelle, Lean, ...).

Proof assistants

Starting from the 70s, people started to develop **proof assistants** which are programs which can check proofs (Agda, Coq, Isabelle, Lean, ...).

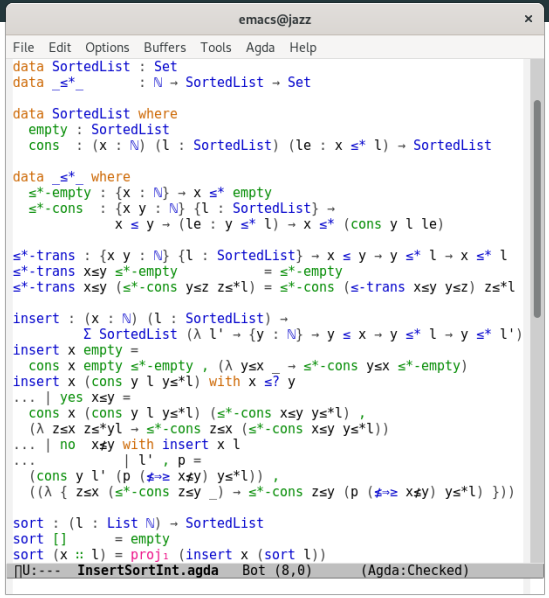
In such a proof assistant, you can

1. express logical formulas
2. gradually prove those formulas
3. extract a program from those proofs

Typical example:

$$\forall l \in (\texttt{List}\ \mathbb{N}).\exists l' \in (\texttt{List}\ \mathbb{N}).(\texttt{Sorted}\ l')$$

Proof assistants



Proof assistants

Those proof assistants guarantee that the program will *always* act according to the specification...

Proof assistants

Those proof assistants guarantee that the program will *always* act according to the specification...

...provided that you believe that those assistants do not have bugs:

- there is a deep well-established theory,
- most proof assistants have a small core,
- part of the proof assistants have been formalized using proof assistants.

Proof assistants

Those proof assistants guarantee that the program will *always* act according to the specification...

...provided that you believe that those assistants do not have bugs:

- there is a deep well-established theory,
- most proof assistants have a small core,
- part of the proof assistants have been formalized using proof assistants.

PS: you also have to trust to compiler

- CompCert: a fully certified compiler

Proof assistants

Those proof assistants guarantee that the program will *always* act according to the specification...

...provided that you believe that those assistants do not have bugs:

- there is a deep well-established theory,
- most proof assistants have a small core,
- part of the proof assistants have been formalized using proof assistants.

PS: you also have to trust to compiler

- CompCert: a fully certified compiler

and the OS,

Proof assistants

Those proof assistants guarantee that the program will *always* act according to the specification...

...provided that you believe that those assistants do not have bugs:

- there is a deep well-established theory,
- most proof assistants have a small core,
- part of the proof assistants have been formalized using proof assistants.

PS: you also have to trust to compiler

- CompCert: a fully certified compiler

and the OS, and the hardware...

Formal methods

Proof assistants are part of **formal methods**, which guarantee behavior of programs.

Formal methods

Proof assistants are part of **formal methods**, which guarantee behavior of programs.

There are various level of automation:

- fully automatic (abstract interpretation, etc.),
- partially automated (Hoare logic, etc.)
- manual (proof assistants, etc.)

In general, more automated means faster but more specific.

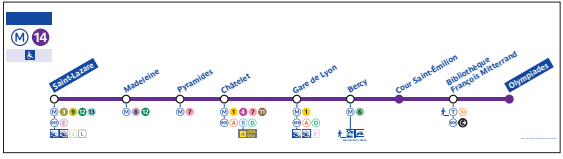
Formal methods

Proof assistants are part of **formal methods**, which guarantee behavior of programs.

They have been successfully used in industry:

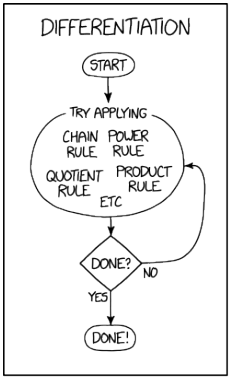
- line 14 and Roissy Val (B-method)
- Airbus
- ...

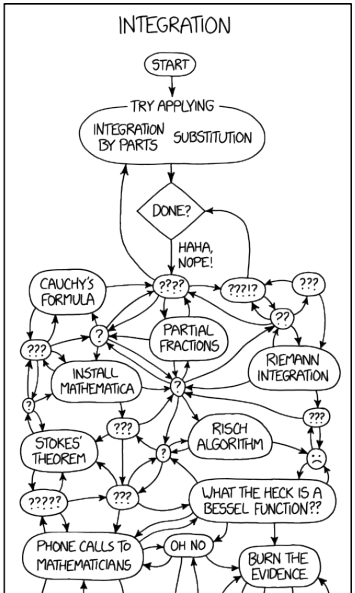
It takes lots of **time** (money), but achieves high level of **guarantee**.



Checking vs proving

proof checking





proving

In this course

The most complicated algorithm that you will prove here is a sorting algorithm.

In this course

The most complicated algorithm that you will prove here is a sorting algorithm.

I can hear you think: “come on, in 2025, we know how to implement sorting”, but

In this course

The most complicated algorithm that you will prove here is a sorting algorithm.

I can hear you think: “come on, in 2025, we know how to implement sorting”, but

- proving more complex programs is “only” a matter of time,

In this course

The most complicated algorithm that you will prove here is a sorting algorithm.

I can hear you think: "come on, in 2025, we know how to implement sorting", but

- proving more complex programs is "only" a matter of time,
- in 2015, a bug was found in the *default* implementation of sorting in *Java*.

Proof checking

Mathematicians are humans and they happen to make mistakes.

For instance,

- 1991: the Fields medalist Voevodsky solves a conjecture of Grothendieck,

  "spaces = strict ∞-categories with weakly invertible morphisms"

Proof checking

Mathematicians are humans and they happen to make mistakes.

For instance,

- 1991: the Fields medalist Voevodsky solves a conjecture of Grothendieck,

  "spaces = strict ∞-categories with weakly invertible morphisms"

- 1998: Simpson finds a counter-example

Proof checking

Mathematicians are humans and they happen to make mistakes.

For instance,

- 1991: the Fields medalist Voevodsky solves a conjecture of Grothendieck,

  "spaces = strict ∞-categories with weakly invertible morphisms"
- 1998: Simpson finds a counter-example
- 2005: Voevodsky gets interested in proof assistants

Proof checking

Mathematicians are humans and they happen to make mistakes.

For instance,

- 1991: the Fields medalist Voevodsky solves a conjecture of Grothendieck,

  "spaces = strict ∞-categories with weakly invertible morphisms"
- 1998: Simpson finds a counter-example
- 2005: Voevodsky gets interested in proof assistants
- 2010: Voevodsky develop new foundations for mathematics

  homotopy type theory

Proof checking

Mathematicians are humans and they happen to make mistakes.

For instance,

- 1991: the Fields medalist Voevodsky solves a conjecture of Grothendieck,

  "spaces = strict ∞-categories with weakly invertible morphisms"
- 1998: Simpson finds a counter-example
- 2005: Voevodsky gets interested in proof assistants
- 2010: Voevodsky develop new foundations for mathematics

  homotopy type theory
- 2013: Voevodsky finally accepts that there is a flaw in his proof

Proof checking

Quoting Voevodsky:

> 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.

Proof checking

Nowadays, in addition to applied mathematics, important mathematical theorems have been formalized:

- the four color theorem[a] (graph theory)
- the Feit-Thompson theorem[b] (group theory)
- the Kepler conjecture[c] (dense sphere packing)
- the liquid tensor experiment[d] (liquid vector spaces)
- the 5th busy beaver number[e] (computability)

[a] https://github.com/coq-community/fourcolor
[b] https://github.com/math-comp/odd-order
[c] https://arxiv.org/abs/1501.02155
[d] https://github.com/leanprover-community/lean-liquid/
[e] https://github.com/ccz181078/Coq-BB5

Proof checking

Nowadays, in addition to applied mathematics, important mathematical theorems have been formalized:

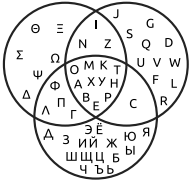
- the independence of the continuum hypothesis [a]
- the existence of sphere eversions[b]
- $\pi_4(\mathbb{S}^3) = \mathbb{Z}_2$[c]

[a] https://flypitch.github.io/
[b] https://github.com/leanprover-community/sphere-eversion
[c] https://github.com/agda/cubical/tree/master/Cubical/Homotopy/Group/Pi4S3

Types as foundations

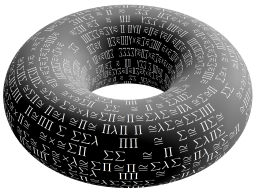
What is mathematics talking about?

- 1901: Russell's paradox in naive set theory
- 1908: Zermelo-Fraenkel set theory
- 1912: Russell's theory of types
- 2013: Voevodsky's homotopy type theory: *type = space*

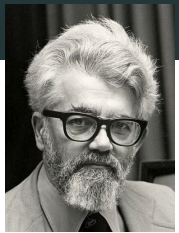


$$
\begin{aligned}
&*54{\cdot}43.\quad \vdash :. \alpha, \beta \in 1 \,.\supset: \alpha \cap \beta = \Lambda \,.\equiv.\, \alpha \cup \beta \in 2 \\
&\quad \textit{Dem.} \\
&\quad \vdash .*54{\cdot}26 \,.\supset \vdash :. \alpha = \iota' x \,.\, \beta = \iota' y \,.\supset: \alpha \cup \beta \in 2 \,.\equiv.\, x \neq y \,. \\
&\quad [*51{\cdot}231] \qquad\qquad \equiv.\, \iota' x \cap \iota' y = \Lambda \,. \\
&\quad [*13{\cdot}12] \qquad\qquad \equiv.\, \alpha \cap \beta = \Lambda \qquad (1) \\
&\quad \vdash .(1) .*11{\cdot}11{\cdot}35 \,.\supset \\
&\qquad \vdash :. (\exists x, y) \,.\, \alpha = \iota' x \,.\, \beta = \iota' y \,.\supset: \alpha \cup \beta \in 2 \,.\equiv.\, \alpha \cap \beta = \Lambda \qquad (2) \\
&\quad \vdash .(2) .*11{\cdot}54 .*52{\cdot}1 \,.\supset \vdash .\ \text{Prop}
\end{aligned}
$$

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$.



Proof searching



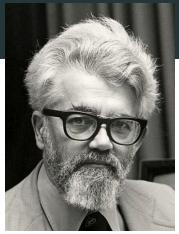
Understanding proof theory allows to

- formulate problems in a logical fashion,
- design new proof search procedures.

In fact, McCarthy, one of the founder of *Artificial Intelligence*, was an advocate of using computational logic in order to represent knowledge and manipulate data.

[Admittedly this is less popular than neural networks today, but one never knows]

Proof searching



Understanding proof theory allows to

- formulate problems in a logical fashion,
- design new proof search procedures.

In fact, McCarthy, one of the founder of *Artificial Intelligence*, was an advocate of using computational logic in order to represent knowledge and manipulate data.

[Admittedly this is less popular than neural networks today, but one never knows]

We don't insist on this here, because these procedures give you little control about the proofs they produce.

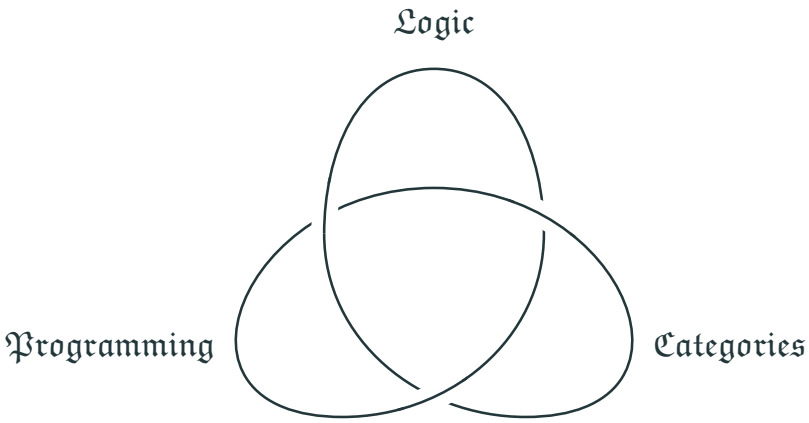
Philosophy

This also provides answer to philosophical / epistemological questions:

- what are the foundations of mathematics?
- what is reasoning?
- what is a proof?
- what does it mean that something exists?
- what does it mean for two things to be equal?
- ...

Religion

Some people base their faith on the **computational trinitarism**:



Part II

Typed functional programming

OCaml in three slides

```
let x = 5
```

OCaml in three slides

```
let x = 5

let f x = 2 * x
```

OCaml in three slides

```
let x = 5

let f x = 2 * x

let () =
  print_string "The result is ";
  print_int (f 5)
```

OCaml in three slides

```
let x = 5

let f x = 2 * x

let () =
  print_string "The result is ";
  print_int (f 5)

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

OCaml in three slides

```
let rec map f l =
  match l with
  | []   -> []
  | x::l -> (f x)::(map f l)
```

OCaml in three slides

```
let rec map f l =
  match l with
  | []   -> []
  | x::l -> (f x)::(map f l)

let () =
  let l1 = [1;2;3] in
  let l2 = map (fun x -> 2*x) l1 in
  assert (l1 = l2)
```

OCaml in three slides

- comparison: = and <> (never == nor !=)

OCaml in three slides

- comparison: `=` and `<>` (never == nor `!=`)
- boolean operations: `&&`, `||`, `not`

OCaml in three slides

- comparison: `=` and `<>` (never `==` nor `!=`)
- boolean operations: `&&`, `||`, `not`
- string concatenation: `s ^ t`

OCaml in three slides

- comparison: `=` and `<>` (never `==` nor `!=`)
- boolean operations: `&&`, `||`, `not`
- string concatenation: `s ^ t`
- patterns always bind:

```
let is_singleton n l =
  match l with
  | [n] -> true
  | _   -> false
```

is not what you think

OCaml in three slides

- comparison: `=` and `<>` (never `==` nor `!=`)
- boolean operations: `&&`, `||`, `not`
- string concatenation: `s ^ t`
- patterns always bind:

```
let is_singleton n l =
  match l with
  | [m] when m = n -> true
  | _              -> false
```

is how you should write it

OCaml in three slides

- comparison: `=` and `<>` (never `==` nor `!=`)
- boolean operations: `&&`, `||`, `not`
- string concatenation: `s ^ t`
- patterns always bind:
- beware of imbricated patterns

```
let is_singleton_singleton l =
  match l with
  | [l'] ->
    match l' with
    | [l''] -> true
    | _     -> false
  | _    -> false
```

is not what your think

OCaml in three slides

- comparison: `=` and `<>` (never `==` nor `!=`)
- boolean operations: `&&`, `||`, `not`
- string concatenation: `s ^ t`
- patterns always bind:
- beware of imbricated patterns

```
let is_singleton_singleton l =
  match l with
  | [l'] ->
    match l' with
    | [l''] -> true
    | _     -> false
    | _     -> false
```

is how OCaml reads it

OCaml in three slides

- comparison: `=` and `<>` (never `==` nor `!=`)
- boolean operations: `&&`, `||`, `not`
- string concatenation: `s ^ t`
- patterns always bind:
- beware of imbricated patterns

```
let is_singleton_singleton l =
  match l with
  | [l'] -> (
      match l' with
      | [l''] -> true
      | _     -> false
    )
  | _    -> false
```

Types in OCaml

Every program has a type:

expression	type
3	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	`'a -> 'a * string`

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	`'a -> 'a * string`
`[3; 4; 1]`	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	`'a -> 'a * string`
`[3; 4; 1]`	`int list`

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	`'a -> 'a * string`
`[3; 4; 1]`	`int list`
`[]`	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	`'a -> 'a * string`
`[3; 4; 1]`	`int list`
`[]`	`'a list`

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	`'a -> 'a * string`
`[3; 4; 1]`	`int list`
`[]`	`'a list`
`List.map`	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	`'a -> 'a * string`
`[3; 4; 1]`	`int list`
`[]`	`'a list`
`List.map`	`'a list -> ('a -> 'b) -> 'b list`

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	`'a -> 'a * string`
`[3; 4; 1]`	`int list`
`[]`	`'a list`
`List.map`	`'a list -> ('a -> 'b) -> 'b list`
`print_string`	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	`'a -> 'a * string`
`[3; 4; 1]`	`int list`
`[]`	`'a list`
`List.map`	`'a list -> ('a -> 'b) -> 'b list`
`print_string`	`string -> unit`

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	`'a -> 'a * string`
`[3; 4; 1]`	`int list`
`[]`	`'a list`
`List.map`	`'a list -> ('a -> 'b) -> 'b list`
`print_string`	`string -> unit`
`fun x -> x x`	

Types in OCaml

Every program has a type:

expression	type
`3`	`int`
`3.0`	`float`
`true`	`bool`
`()`	`unit`
`fun x -> 2 * x`	`int -> int`
`fun x -> (2 * x, "x")`	`int -> int * string`
`fun x -> (x, "x")`	`'a -> 'a * string`
`[3; 4; 1]`	`int list`
`[]`	`'a list`
`List.map`	`'a list -> ('a -> 'b) -> 'b list`
`print_string`	`string -> unit`
`fun x -> x x`	`Error: This expression has type 'a -> 'b but`

Types in OCaml

Typing guarantees that functions will always get arguments as expected.

The following will be rejected:

```
3 * "x"
a.(2.1)
```

⋮

Types in OCaml

Typing guarantees that functions will always get arguments as expected.

The following will be rejected:

```
3 * "x"
a.(2.1)
   ⋮
```

but also

```
"abc" ^ 3
4 + 2.1
   ⋮
```

More on this later.

Types in OCaml

Types in OCaml are

Types in OCaml

Types in OCaml are

- **static**: checked at compilation time,

Types in OCaml

Types in OCaml are

- **static**: checked at compilation time,
- **inferred**: no type annotation is required, the compiler "guesses" the type,

Types in OCaml

Types in OCaml are

- **static**: checked at compilation time,
- **inferred**: no type annotation is required, the compiler "guesses" the type,
- **polymorphic**: a type such as

  ```
  'a -> 'a
  ```

  is implicitly universally quantified on `'a`,

Types in OCaml

Types in OCaml are

- **static**: checked at compilation time,
- **inferred**: no type annotation is required, the compiler "guesses" the type,
- **polymorphic**: a type such as

  ```
  'a -> 'a
  ```

  is implicitly universally quantified on `'a`,
- **principal**: the most general type is always inferred

  ```
  fun x -> x : 'a -> 'a
  ```

  but we can specify types if we want

  ```
  fun (x : int) -> x : int -> int
  ```

Recursive types

Values of recursive types can be observed by **pattern matching**.

For instance, on lists:

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

Recursive types

We can also define custom **recursive types**:

```
type ilist =
  | Nil
  | Cons of int * ilist
```

A typical value is

Recursive types

We can also define custom **recursive types**:

```
type ilist =
  | Nil
  | Cons of int * ilist
```

A typical value is

```
Nil
```

Recursive types

We can also define custom **recursive types**:

```
type ilist =
  | Nil
  | Cons of int * ilist
```

A typical value is

```
                    Nil
Cons (3 , Nil)
```

Recursive types

We can also define custom **recursive types**:

```
type ilist =
  | Nil
  | Cons of int * ilist
```

A typical value is

```
                                  Nil
                        Cons (3 , Nil)
              Cons (5 , Cons (3 , Nil))
```

Recursive types

We can also define custom **recursive types**:

```
type ilist =
  | Nil
  | Cons of int * ilist
```

A typical value is

```
                                Nil
                        Cons (3 , Nil)
              Cons (5 , Cons (3 , Nil))
                        ⋮
```

Recursive types

We can also define custom **parametrized recursive types**:

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

A typical value is

```
                                Nil
                        Cons (3 , Nil)
              Cons (5 , Cons (3 , Nil))
                        ⋮
```

Recursive types

We can also define custom **parametrized recursive types**:

```
type 'a list =
  | []
  | 'a :: 'a list
```

A typical value is

```
                                Nil
                    Cons (3 , Nil)
Cons (5 , Cons (3 , Nil))
                    ⋮
```

Recursive types

Functions on recursive types are typically defined by recurrence:

```
let rec length l =
  match l with
  | Nil           -> 0
  | Cons (x , l') -> 1 + length l'
```

Recursive types

Functions on recursive types are typically defined by recurrence:

```
let rec concat l m =
  match l with
  | Nil           -> m
  | Cons (x , l') -> Cons (x , concat l' m)
```

Recursive types

Many usual types can be defined as recursive types.

Booleans:

Conditional `if b then e1 else e2` can be written as

Recursive types

Many usual types can be defined as recursive types.

Booleans:

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

Conditional `if b then e1 else e2` can be written as

Recursive types

Many usual types can be defined as recursive types.

Booleans:

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

Conditional `if b then e1 else e2` can be written as

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

Recursive types

Many usual types can be defined as recursive types.

Natural numbers (in unary notation):

Addition can be computed as

Recursive types

Many usual types can be defined as recursive types.

Natural numbers (in unary notation):

```
type nat =
  | Z
  | S of nat
```

Addition can be computed as

Recursive types

Many usual types can be defined as recursive types.

Natural numbers (in unary notation):

```
type nat =
  | Z
  | S of nat
```

Addition can be computed as

```
let rec add x y =
  match x with
  | Z    -> y
  | S x' -> S (add x' y)
```

Recursive types

Many usual types can be defined as recursive types.

Unit:

A function `let f () = e` can be written as

Recursive types

Many usual types can be defined as recursive types.

Unit:

```
type unit =
  | T
```

which is usually written `()` instead of `T`.

A function `let f () = e` can be written as

Recursive types

Many usual types can be defined as recursive types.

Unit:

```
type unit =
  | T
```

which is usually written `()` instead of `T`.

A function `let f () = e` can be written as

```
let f x =
  match x with
  | T -> e
```

Other types

There are other ways of building types:

- products:

`(3 , "x")` is of type `int * string`

Other types

There are other ways of building types:

- products:

  `(3 , "x")` is of type `int * string`

- records, objects, etc.

OCaml in practice

In practice, you should use **Emacs** to edit ml files.

The only shortcut you need to know is

`C-c C-e`

which evaluates the current function
(the first time you also need to press enter to launch `ocaml`)

OCaml in practice

In case you want to use **VS Code**, you need to select the code and type

`shift-enter`

Part III

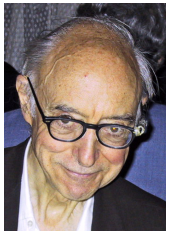
The proof-as-program correspondence

Types as formulas

There is a wonderful thing, called the **proof-as-program correspondence**, due to

Curry

and



Howard

which states that

a type is the same as a formula

and

a program is the same as a proof.

Arrow as implication

Given types `T` and `U`, the type

```
T -> U
```

- the type of functions from `T` to `U`,

Think of `string_of_int` of type `int -> string`.

Arrow as implication

Given types `T` and `U`, the type

```
T -> U
```

- the type of functions from `T` to `U`,
- the type of programs which transform a `T` into a `U`,

Think of `string_of_int` of type `int -> string`.

Arrow as implication

Given types `T` and `U`, the type

```
T -> U
```

- the type of functions from `T` to `U`,
- the type of programs which transform a `T` into a `U`,
- the type of programs thanks to which having a `T` implies having a `U`,

Think of `string_of_int` of type `int -> string`.

Arrow as implication

Given types `T` and `U`, the type

```
T -> U
```

- the type of functions from `T` to `U`,
- the type of programs which transform a `T` into a `U`,
- the type of programs thanks to which having a `T` implies having a `U`,
- the formula $T \Rightarrow U$,

Think of `string_of_int` of type `int -> string`.

Arrow as implication

Given types `T` and `U`, the type

```
T -> U
```

- the type of functions from `T` to `U`,
- the type of programs which transform a `T` into a `U`,
- the type of programs thanks to which having a `T` implies having a `U`,
- the formula $T \Rightarrow U$,
- the type of proofs of $T \Rightarrow U$.

Think of `string_of_int` of type `int -> string`.

Arrow as implication

We can thus think of a program of type

```
'a -> 'a
```

as a proof of

$$A \Rightarrow A$$

Arrow as implication

We can thus think of a program of type

```
'a -> 'a
```

as a proof of

$$A \Rightarrow A$$

It can be proved by

Arrow as implication

We can thus think of a program of type

```
'a -> 'a
```

as a proof of

$$A \Rightarrow A$$

It can be proved by

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

Arrow as implication

The formula

$$A \Rightarrow B \Rightarrow A$$

Arrow as implication

The formula

$$A \Rightarrow B \Rightarrow A$$

i.e. the type

```
'a -> 'b -> 'a
```

Arrow as implication

The formula

$$A \Rightarrow B \Rightarrow A$$

i.e. the type

```
'a -> 'b -> 'a
```

can be proved by

Arrow as implication

The formula

$$A \Rightarrow B \Rightarrow A$$

i.e. the type

```
'a -> 'b -> 'a
```

can be proved by

```
let k = fun x y -> x
```

Arrow as implication

The formula

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

Arrow as implication

The formula

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

i.e. the type

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

Arrow as implication

The formula

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

i.e. the type

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

can be proved by

Arrow as implication

The formula

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

i.e. the type

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

can be proved by

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

The proof-as-program correspondence

We will make the following precise:

Theorem
A formula can be proved (for a suitable notion of provability) if and only there is a program of the corresponding type (for a suitable subset of OCaml).

In other words,

$$\text{PROGRAM} = \text{PROOF}$$

(at least for existence)

Conjunction

The correspondence would be boring if it was limited to implication.

The formula

$$A \wedge B$$

can be interpreted as the type

Conjunction

The correspondence would be boring if it was limited to implication.

The formula

$$A \wedge B$$

can be interpreted as the type

```
'a * 'b
```

Conjunction

The formula

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

Conjunction

The formula

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

corresponding to the type

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

Conjunction

The formula

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

corresponding to the type

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

can be proved by

Conjunction

The formula

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

corresponding to the type

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

can be proved by

```
let proj1 = fun (x , y) -> x
```

Conjunction

The formula

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

Conjunction

The formula

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

corresponding to the type

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

Conjunction

The formula

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

corresponding to the type

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

can be proved by

Conjunction

The formula

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

corresponding to the type

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

can be proved by

```
let comm = fun (x , y) -> (y , x)
```

Truth

The truth formula

$$\top$$

can be interpreted as the type

Truth

The truth formula

$$\top$$

can be interpreted as the type

```
unit
```

whose only value is

```
()
```

Truth

The formula

$$A \Rightarrow \top$$

Truth

The formula

$$A \Rightarrow \top$$

corresponding to the type

```
'a -> unit
```

Truth

The formula

$$A \Rightarrow \top$$

corresponding to the type

```
'a -> unit
```

can be proved by

Truth

The formula

$$A \Rightarrow \top$$

corresponding to the type

```
'a -> unit
```

can be proved by

```
fun x -> ()
```

Falsity

The falsity formula

$$\bot$$

can be interpreted as the type

Falsity

The falsity formula

$$\bot$$

can be interpreted as the type

```
type empty = |
```

which is a recursive type with no constructor.

Falsity

The formula

$$\bot \Rightarrow A$$

Falsity

The formula

$$\bot \Rightarrow A$$

corresponding to the type

```
empty -> 'a
```

Falsity

The formula

$$\bot \Rightarrow A$$

corresponding to the type

```
empty -> 'a
```

can be proved by

Falsity

The formula

$$\bot \Rightarrow A$$

corresponding to the type

```
empty -> 'a
```

can be proved by

```
let absurd = fun x -> match x with _ -> .
```

Negation

As usual, negation can be defined as

$$\neg A \quad =$$

Negation

As usual, negation can be defined as

$$\neg A \quad = \quad A \Rightarrow \bot$$

which corresponds to the type

```
'a -> empty
```

Negation

The *contraposition* formula

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

Negation

The *contraposition* formula

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

corresponding to the type

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

Negation

The *contraposition* formula

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

corresponding to the type

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

can be proved by

Negation

The *contraposition* formula

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

corresponding to the type

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

can be proved by

```
let contr = fun f k a -> k (f a)
```

Disjunction

The disjunction

$$A \vee B$$

can be interpreted as the recursive type

Disjunction

The disjunction

$$A \vee B$$

can be interpreted as the recursive type

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

Disjunction

The disjunction

$$A \vee B$$

can be interpreted as the recursive type

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

(this is `('a , 'b) Either.t` in the standard library)

Disjunction

The distributivity formula

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

Disjunction

The distributivity formula

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

corresponding to the type

```
('a * ('b , 'c) coprod) -> ('a * 'b , 'a * 'c) coprod
```

Disjunction

The distributivity formula

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

corresponding to the type

```
('a * ('b , 'c) coprod) -> ('a * 'b , 'a * 'c) coprod
```

can be proved by

Disjunction

The distributivity formula

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

corresponding to the type

```
('a * ('b , 'c) coprod) -> ('a * 'b , 'a * 'c) coprod
```

can be proved by

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

Corner cases

The correspondence between

- a formula is provable,
- there exists a program of the corresponding type

is not perfect because

- OCaml is not intended for that, and
- we need to do more theory.

For languages such as Agda, the match is perfect.

Corner cases: too many provable formulas

We can prove absurd formulas such as

$$A \Rightarrow B$$

by

Corner cases: too many provable formulas

We can prove absurd formulas such as

$$A \Rightarrow B$$

by

- using side effects:

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

Corner cases: too many provable formulas

We can prove absurd formulas such as

$$A \Rightarrow B$$

by

- using side effects:

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

- using non-termination:

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

Corner cases: too many provable formulas

We can prove absurd formulas such as

$$A \Rightarrow B$$

by

- using side effects:

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

- using non-termination:

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

From which we can "prove" pretty much everything:

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

Corner cases: too few provable formulas

The formula

$$A \vee \neg A$$

corresponding to the type

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

Corner cases: too few provable formulas

The formula

$$A \vee \neg A$$

corresponding to the type

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

has no simple proof.

Part IV

Typed programs are safe

The usefulness of typing

As Milner put it:

> *Well-typed programs cannot go wrong.*

The usefulness of typing

As Milner put it:

> *Well-typed programs cannot go wrong.*

We will see that typed programs are **safe**:

The usefulness of typing

As Milner put it:

> *Well-typed programs cannot go wrong.*

We will see that typed programs are **safe**:

- **subject reduction**: typing is preserved along reduction,

The usefulness of typing

As Milner put it:

> *Well-typed programs cannot go wrong.*

We will see that typed programs are **safe**:

- **subject reduction**: typing is preserved along reduction,
- **progress**: a program which is not a value always reduces, i.e. execution is never stuck

The usefulness of typing

As Milner put it:

> *Well-typed programs cannot go wrong.*

We will see that typed programs are **safe**:

- **subject reduction**: typing is preserved along reduction,
- **progress**: a program which is not a value always reduces,
  i.e. execution is never stuck

In order to formalize this, we need to define the **reduction** and **typing** of our language.

A simple programming language

A **program** is a term generated by

p, q	$::=$	b	a boolean $b \in \{\texttt{true}, \texttt{false}\}$
	$\mid$	n	an integer $n \in \mathbb{Z}$
	$\mid$	p `+` q	an addition
	$\mid$	p `<` q	a comparison
	$\mid$	`if` p `then` q `else` r	a branching

A simple programming language

A **program** is a term generated by

p, q	$::=$	b	a boolean $b \in \{\texttt{true}, \texttt{false}\}$
	$\mid$	n	an integer $n \in \mathbb{Z}$
	$\mid$	p `+` q	an addition
	$\mid$	p `<` q	a comparison
	$\mid$	`if` p `then` q `else` r	a branching

Note that we can have programs of the form `3 + true`!

A simple programming language

A **program** is a term generated by

p, q	$::=$	b	a boolean $b \in \{\texttt{true}, \texttt{false}\}$
	$\mid$	n	an integer $n \in \mathbb{Z}$
	$\mid$	p `+` q	an addition
	$\mid$	p `<` q	a comparison
	$\mid$	`if` p `then` q `else` r	a branching

Note that we can have programs of the form `3 + true`!

A **value** is a program which is either a boolean b or an integer n.

Typing

A **type** is either `bool` or `int`.

Typing

A **type** is either `bool` or `int`.

The fact that p has type A is written

$$\vdash p : A$$

Typing

A **type** is either `bool` or `int`.

The fact that p has type A is written

$$\vdash p : A$$

The **typing rules** are

$$\frac{}{\vdash n : \texttt{int}} \qquad \frac{}{\vdash b : \texttt{bool}}$$

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

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

Typing

A program p **has type** A when $\vdash p : A$ can be derived.

For instance:

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

This is called a **derivation tree** and we can reason on it.

Type uniquenes

Theorem (Uniqueness of typing)
If $\vdash p : A$ *and* $\vdash p : B$ *are derivable then* $A = B$.

Type uniquenes

Theorem (Uniqueness of typing)
If $\vdash p : A$ *and* $\vdash p : B$ *are derivable then* $A = B$.

Proof.
By induction on p (note that there is at most one rule for each form of program):

- if p is of the form $p_1 \ + \ p_2$ then necessarily the last typing rule is

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

- ...

□

Type uniquenes

Theorem (Uniqueness of typing)
If $\vdash p : A$ *and* $\vdash p : B$ *are derivable then* $A = B$.

Note: this does not hold in OCaml since

Type uniquenes

Theorem (Uniqueness of typing)
If $\vdash p : A$ *and* $\vdash p : B$ *are derivable then* $A = B$.

Note: this does not hold in OCaml since

```
fun x -> x
```

has types

```
'a -> 'a        int -> int        string -> string
```

but there is a most general one (`'a -> 'a`), which is the inferred type.

Reduction

The **reduction** relation $p \longrightarrow q$ describes when a program p evaluates to q in one step.

Reduction

The **reduction** relation $p \longrightarrow q$ describes when a program p evaluates to q in one step.

It is the smallest relation such that

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

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

For instance:

- $\texttt{3 + 2} \longrightarrow 5$
- $\texttt{(6 + 3) + (1 + 1)} \longrightarrow \texttt{9 + (1 + 1)}$
- $\texttt{(6 + 3) + (1 + 1)} \longrightarrow \texttt{(6 + 3) + 2}$

Reduction

The **reduction** relation $p \longrightarrow q$ describes when a program p evaluates to q in one step.

It is the smallest relation such that

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

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

For instance:

$$\texttt{(2 + 2) < 3} \longrightarrow \texttt{4 < 3} \longrightarrow \texttt{false}$$

Reduction

The **reduction** relation $p \longrightarrow q$ describes when a program p evaluates to q in one step.

It is the smallest relation such that

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

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

For instance:

$$\begin{aligned} \texttt{if 2 < 3 then 5 + 1 else 9} &\longrightarrow \texttt{if true then 5 + 1 else 9} \\ &\longrightarrow \texttt{5 + 1} \\ &\longrightarrow \texttt{6} \end{aligned}$$

Irreducible programs

We have to closely related notions:

- **values** are either booleans or integers,
- **irreducible** programs are programs that cannot reduce.

Irreducible programs

We have to closely related notions:

- **values** are either booleans or integers,
- **irreducible** programs are programs that cannot reduce.

Note that values are irreducible:

$$3 \not\longrightarrow \ldots \qquad\qquad \texttt{true} \not\longrightarrow \ldots$$

Irreducible programs

We have to closely related notions:

- **values** are either booleans or integers,
- **irreducible** programs are programs that cannot reduce.

Note that values are irreducible:

$$3 \not\longrightarrow \ldots \qquad\qquad \texttt{true} \not\longrightarrow \ldots$$

but there are irreducible programs which are not values:

Irreducible programs

We have to closely related notions:

- **values** are either booleans or integers,
- **irreducible** programs are programs that cannot reduce.

Note that values are irreducible:

$$3 \not\longrightarrow \ldots \qquad\qquad \texttt{true} \not\longrightarrow \ldots$$

but there are irreducible programs which are not values:

- `3 + true`
- `if 5 then` p `else` q
- $\ldots$

Those correspond to erroneous programs.

Irreducible programs

We have to closely related notions:

- **values** are either booleans or integers,
- **irreducible** programs are programs that cannot reduce.

Note that values are irreducible:

$$3 \not\longrightarrow \ldots \qquad\qquad \texttt{true} \not\longrightarrow \ldots$$

but there are irreducible programs which are not values:

- `3 + true`
- `if 5 then` p `else` q
- $\ldots$

Those correspond to erroneous programs.

We will prove that for typable programs, the two notions coincide!

Safety: subject reduction

Theorem (Subject reduction)
If $p \longrightarrow p'$ and p has type A then p' also has type A.

Safety: subject reduction

Theorem (Subject reduction)
If $p \longrightarrow p'$ and p has type A then p' also has type A.

Proof.
By induction on the derivation of $p \longrightarrow p'$.

Safety: subject reduction

Theorem (Subject reduction)
If $p \longrightarrow p'$ and p has type A then p' also has type A.

Proof.
By induction on the derivation of $p \longrightarrow p'$.

If the last rule is

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

Safety: subject reduction

Theorem (Subject reduction)
If $p \longrightarrow p'$ and p has type A then p' also has type A.

Proof.
By induction on the derivation of $p \longrightarrow p'$.

If the last rule is

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

then the derivation of $\vdash p : A$
necessarily ends with

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

Safety: subject reduction

Theorem (Subject reduction)
If $p \longrightarrow p'$ and p has type A then p' also has type A.

Proof.
By induction on the derivation of $p \longrightarrow p'$.

If the last rule is

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

thus $\vdash p_1 : \texttt{int}$ is derivable,

then the derivation of $\vdash p : A$
necessarily ends with

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

Safety: subject reduction

Theorem (Subject reduction)
If $p \longrightarrow p'$ and p has type A then p' also has type A.

Proof.
By induction on the derivation of $p \longrightarrow p'$.

If the last rule is

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

then the derivation of $\vdash p : A$
necessarily ends with

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

thus $\vdash p_1 : \texttt{int}$ is derivable,
thus $\vdash p_1' : \texttt{int}$ is derivable by induction hypothesis,

Safety: subject reduction

Theorem (Subject reduction)
If $p \longrightarrow p'$ and p has type A then p' also has type A.

Proof.
By induction on the derivation of $p \longrightarrow p'$.

If the last rule is

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

then the derivation of $\vdash p : A$ necessarily ends with

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

thus $\vdash p_1 : \texttt{int}$ is derivable,
thus $\vdash p_1' : \texttt{int}$ is derivable by induction hypothesis,
thus $\vdash p_1' \ + \ p_2 : \texttt{int}$ is derivable:

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

Safety: subject reduction

Theorem (Subject reduction)
If $p \longrightarrow p'$ and p has type A then p' also has type A.

Proof.
By induction on the derivation of $p \longrightarrow p'$.

If the last rule is

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

then the derivation of $\vdash p : A$ necessarily ends with

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

thus $\vdash p_1 : \texttt{int}$ is derivable,
thus $\vdash p_1' : \texttt{int}$ is derivable by induction hypothesis,
thus $\vdash p_1' \ + \ p_2 : \texttt{int}$ is derivable:

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

Other cases are similar. □

Safety: progress

Theorem (Progress)
If p is a program of type A which is not value then p reduces (to some p').

Safety: progress

Theorem (Progress)
If p is a program of type A which is not value then p reduces (to some p').

Proof.
By induction on the derivation of $\vdash p : A$.

Safety: progress

Theorem (Progress)
If p is a program of type A which is not value then p reduces (to some p').

Proof.
By induction on the derivation of $\vdash p : A$.

Suppose that the last rule is

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

Safety: progress

Theorem (Progress)
If p is a program of type A which is not value then p reduces (to some p').

Proof.
By induction on the derivation of $\vdash p : A$.

Suppose that the last rule is

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

If p_1 is not a value then, by induction hypothesis, $p_1 \longrightarrow p_1'$ and thus $p_1 \ \texttt{+} \ p_2 \longrightarrow p_1' \ \texttt{+} \ p_2$.

Safety: progress

Theorem (Progress)
If p is a program of type A which is not value then p reduces (to some p').

Proof.
By induction on the derivation of $\vdash p : A$.

Suppose that the last rule is

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

If p_1 is not a value then, by induction hypothesis, $p_1 \longrightarrow p_1'$ and thus $p_1 \ \texttt{+} \ p_2 \longrightarrow p_1' \ \texttt{+} \ p_2$.

If p_2 is not a value then, by induction hypothesis, $p_2 \longrightarrow p_2'$ and thus $p_1 \ \texttt{+} \ p_2 \longrightarrow p_1 \ \texttt{+} \ p_2'$.

Safety: progress

Theorem (Progress)
If p is a program of type A which is not value then p reduces (to some p').

Proof.
By induction on the derivation of $\vdash p : A$.

Suppose that the last rule is

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

If p_1 is not a value then, by induction hypothesis, $p_1 \longrightarrow p_1'$ and thus $p_1 \ + \ p_2 \longrightarrow p_1' \ + \ p_2$.

If p_2 is not a value then, by induction hypothesis, $p_2 \longrightarrow p_2'$ and thus $p_1 \ + \ p_2 \longrightarrow p_1 \ + \ p_2'$.

If both p_1 and p_2 are values then they are necessarily integers and thus $p_1 \ + \ p_2 \longrightarrow n$.

Safety: progress

Theorem (Progress)
If p is a program of type A which is not value then p reduces (to some p').

Proof.
By induction on the derivation of $\vdash p : A$.

Suppose that the last rule is

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

If p_1 is not a value then, by induction hypothesis, $p_1 \longrightarrow p_1'$ and thus $p_1 \ + \ p_2 \longrightarrow p_1' \ + \ p_2$.

If p_2 is not a value then, by induction hypothesis, $p_2 \longrightarrow p_2'$ and thus $p_1 \ + \ p_2 \longrightarrow p_1 \ + \ p_2'$.

If both p_1 and p_2 are values then they are necessarily integers and thus $p_1 \ + \ p_2 \longrightarrow n$.

Other cases are similar. □

Safety

Theorem (Safety)
Given a program p of type A, either

- *there is an infinite sequence of reductions:* $p \longrightarrow p_1 \longrightarrow p_2 \longrightarrow \ldots$,
- *or the reduction ends on a value:* $p \longrightarrow p_1 \longrightarrow p_2 \longrightarrow \ldots \longrightarrow p_n = v$.

Safety

Theorem (Safety)
Given a program p of type A, either

- *there is an infinite sequence of reductions:* $p \longrightarrow p_1 \longrightarrow p_2 \longrightarrow \ldots$,
- *or the reduction ends on a value:* $p \longrightarrow p_1 \longrightarrow p_2 \longrightarrow \ldots \longrightarrow p_n = v$.

Proof.
Suppose given a finite sequence of reductions from p,

$$p \longrightarrow p_1 \longrightarrow \ldots \longrightarrow p_n$$

such that all the p_i are of type A. Then, by progress, either

- p_n is a value, or
- $p_n \longrightarrow p_{n+1}$ and p_{n+1} is of type A by subject reduction. □

Safety

The typing system thus ensures that we will **never get stuck** because we have the wrong data:

```
true + 3
```

Safety

The typing system thus ensures that we will **never get stuck** because we have the wrong data:

```
true + 3
```

We are on the **safe** side, but we reject legit programs:

```
if true then 3 else false
```

Safety

The typing system thus ensures that we will **never get stuck** because we have the wrong data:

```
true + 3
```

We are on the **safe** side, but we reject legit programs:

```
if true then 3 else false
```

We only prevent data errors, but **not all errors**: the program

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

should be given the type

Safety

The typing system thus ensures that we will **never get stuck** because we have the wrong data:

```
true + 3
```

We are on the **safe** side, but we reject legit programs:

```
if true then 3 else false
```

We only prevent data errors, but **not all errors**: the program

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

should be given the type

$$\{n : \texttt{int} \mid n \neq 2\} \texttt{ -> int}$$

Part V

Abstract definition of recursive types

Recursive definitions

We write $\mathcal{U}$ for a set, which we think of as all possible OCaml values.

We write $\mathcal{P}(\mathcal{U})$ for the set of subsets of $\mathcal{U}$.

Recursive definitions

We write $\mathcal{U}$ for a set, which we think of as all possible OCaml values.

We write $\mathcal{P}(\mathcal{U})$ for the set of subsets of $\mathcal{U}$.

Given $F : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ we say that $X \in \mathcal{P}(\mathcal{U})$ is a

- **prefixpoint** when $F(X) \subseteq X$,
- **fixpoint** when $F(X) = X$.

Recursive definitions

Consider the following type for binary trees labeled by integers:

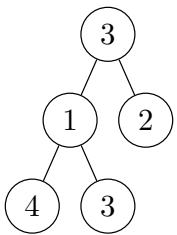
```
type tree =
  | Leaf of int
  | Node of int * tree * tree
```

Recursive definitions

Consider the following type for binary trees labeled by integers:

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

For instance, we have the tree




```
Node (3, Node (1, Leaf 4, Leaf 3), Leaf 2)
```

Recursive definitions

Consider the following type for binary trees labeled by integers:

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

This type induces a function $F : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ defined by

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

Recursive definitions

Consider the following type for binary trees labeled by integers:

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

This type induces a function $F : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ defined by

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

and the set $\mathcal{T}$ of trees is the smallest subset of $\mathcal{U}$ such that

$$F(\mathcal{T}) \subseteq \mathcal{T}$$

Recursive definitions

Consider the following type for binary trees labeled by integers:

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

This type induces a function $F : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ defined by

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

and the set $\mathcal{T}$ of trees is the smallest subset of $\mathcal{U}$ such that

$$F(\mathcal{T}) \subseteq \mathcal{T}$$

It turns out that we actually have $F(\mathcal{T}) = \mathcal{T}$ and it is the smallest such.

Recursive definitions

First note that the function

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

is increasing:

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

Recursive definitions

First note that the function

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

is increasing:

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

We will see that for such a function

- the smallest prefixpoint exists, and
- it coincides with the smallest fixpoint.

The Knaster-Tarski theorem

Theorem
Given an increasing $F : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$*, 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* $\mathrm{fix}(F) \subseteq X$ *for every fixpoint* X *of* F*.*

The Knaster-Tarski theorem: proof

Proof.

We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$.

The Knaster-Tarski theorem: proof

Proof.

We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$.

Given $X \in \mathcal{C}$, we have $\mathrm{fix}(F) = \bigcap \mathcal{C} \subseteq X$ (*)

The Knaster-Tarski theorem: proof

Proof.

We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$.

Given $X \in \mathcal{C}$, we have $\mathrm{fix}(F) = \bigcap \mathcal{C} \subseteq X$ (*)

and therefore, since F is increasing, $F(\mathrm{fix}(F)) \subseteq F(X) \subseteq X$ (**)

The Knaster-Tarski theorem: proof

Proof.

We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$.

Given $X \in \mathcal{C}$, we have $\mathrm{fix}(F) = \bigcap \mathcal{C} \subseteq X$ (*)

and therefore, since F is increasing, $F(\mathrm{fix}(F)) \subseteq F(X) \subseteq X$ (**)

from which we deduce $F(\mathrm{fix}(F)) \subseteq \bigcap \mathcal{C} = \mathrm{fix}(F)$.

The Knaster-Tarski theorem: proof

Proof.

We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$.

Given $X \in \mathcal{C}$, we have $\text{fix}(F) = \bigcap \mathcal{C} \subseteq X$ (*)

and therefore, since F is increasing, $F(\text{fix}(F)) \subseteq F(X) \subseteq X$ (**)

from which we deduce $F(\text{fix}(F)) \subseteq \bigcap \mathcal{C} = \text{fix}(F)$.

Moreover, by monotonicity again, we have $F(F(\text{fix}(F))) \subseteq F(\text{fix}(F))$

The Knaster-Tarski theorem: proof

Proof.

We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$.

Given $X \in \mathcal{C}$, we have $\text{fix}(F) = \bigcap \mathcal{C} \subseteq X$ (*)

and therefore, since F is increasing, $F(\text{fix}(F)) \subseteq F(X) \subseteq X$ (**)

from which we deduce $F(\text{fix}(F)) \subseteq \bigcap \mathcal{C} = \text{fix}(F)$.

Moreover, by monotonicity again, we have $F(F(\text{fix}(F))) \subseteq F(\text{fix}(F))$

therefore, $F(\text{fix}(F)) \in \mathcal{C}$,

The Knaster-Tarski theorem: proof

Proof.

We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$.

Given $X \in \mathcal{C}$, we have $\text{fix}(F) = \bigcap \mathcal{C} \subseteq X$ (*)

and therefore, since F is increasing, $F(\text{fix}(F)) \subseteq F(X) \subseteq X$ (**)

from which we deduce $F(\text{fix}(F)) \subseteq \bigcap \mathcal{C} = \text{fix}(F)$.

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

The Knaster-Tarski theorem: proof

Proof.

We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$.

Given $X \in \mathcal{C}$, we have $\text{fix}(F) = \bigcap \mathcal{C} \subseteq X$ (*)

and therefore, since F is increasing, $F(\text{fix}(F)) \subseteq F(X) \subseteq X$ (**)

from which we deduce $F(\text{fix}(F)) \subseteq \bigcap \mathcal{C} = \text{fix}(F)$.

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

We have shown that $\text{fix}(F)$ is a fixpoint of F.

The Knaster-Tarski theorem: proof

Proof.

We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$.

Given $X \in \mathcal{C}$, we have $\text{fix}(F) = \bigcap \mathcal{C} \subseteq X$ (*)

and therefore, since F is increasing, $F(\text{fix}(F)) \subseteq F(X) \subseteq X$ (**)

from which we deduce $F(\text{fix}(F)) \subseteq \bigcap \mathcal{C} = \text{fix}(F)$.

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

We have shown that $\text{fix}(F)$ is a fixpoint of F.

Given a fixpoint X of F, we have $X \in \mathcal{C}$

The Knaster-Tarski theorem: proof

Proof.

We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$.

Given $X \in \mathcal{C}$, we have $\text{fix}(F) = \bigcap \mathcal{C} \subseteq X$ (*)

and therefore, since F is increasing, $F(\text{fix}(F)) \subseteq F(X) \subseteq X$ (**)

from which we deduce $F(\text{fix}(F)) \subseteq \bigcap \mathcal{C} = \text{fix}(F)$.

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

We have shown that $\text{fix}(F)$ is a fixpoint of F.

Given a fixpoint X of F, we have $X \in \mathcal{C}$

thus, by (**), $\text{fix}(F) = F(\text{fix}(F)) \subseteq X$

The Knaster-Tarski theorem: proof

Proof.

We write $\mathcal{C} = \{X \in \mathcal{P}(\mathcal{U}) \mid F(X) \subseteq X\}$.

Given $X \in \mathcal{C}$, we have $\text{fix}(F) = \bigcap \mathcal{C} \subseteq X$ (*)

and therefore, since F is increasing, $F(\text{fix}(F)) \subseteq F(X) \subseteq X$ (**)

from which we deduce $F(\text{fix}(F)) \subseteq \bigcap \mathcal{C} = \text{fix}(F)$.

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

We have shown that $\text{fix}(F)$ is a fixpoint of F.

Given a fixpoint X of F, we have $X \in \mathcal{C}$

thus, by (**), $\text{fix}(F) = F(\text{fix}(F)) \subseteq X$

i.e. $\text{fix}(F)$ is the smallest fixpoint. □

Recursive definitions

Given a recursive type inducing a function F, we can think of $\mathrm{fix}(F)$ as its set of elements.

The Knaster-Tarski theorem

Note that the Knaster-Tarski theorem generalizes to any complete semilattice.

The Kleene fixpoint theorem

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

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

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.

Induction on recursive types

As a direct corollary,

Theorem (Induction principle)
Given a set X *such that* $F(X) \subseteq X$*, we have* $\mathrm{fix}(F) \subseteq X$*.*

This is abstractly an induction principle on types.

Induction on recursive types

Consider the type of natural numbers

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

Induction on recursive types

Consider the type of natural numbers

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

Its associated function is

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

Induction on recursive types

Consider the type of natural numbers

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

Its associated function is

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

and its smallest fixpoint is

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

Induction on recursive types

Consider the type of natural numbers

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

Its associated function is

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

and its smallest fixpoint is

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

Given a property $P(n)$, consider the set $X = \{n \in \mathbb{N} \mid P(n)\}$.

Induction on recursive types

Consider the type of natural numbers

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

Its associated function is

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

and its smallest fixpoint is

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

Given a property $P(n)$, consider the set $X = \{n \in \mathbb{N} \mid P(n)\}$.
We have $F(X) \subseteq X$ if and only if $P(0)$, and $P(n)$ implies $P(S\,n)$.

Induction on recursive types

Consider the type of natural numbers

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

Its associated function is

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

and its smallest fixpoint is

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

Given a property $P(n)$, consider the set $X = \{n \in \mathbb{N} \mid P(n)\}$.
We have $F(X) \subseteq X$ if and only if $P(0)$, and $P(n)$ implies $P(S\,n)$.
The induction principle, is thus the usual recurrence principle:

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

Induction on recursive types

Consider the type `empty`. We have $F(X) = \emptyset$ and thus $\mathrm{fix}(F) = \emptyset$. The induction principle states

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

An example of a non-smallest fixpoint

Note that the function

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

admits a non-smallest fixpoint:

$$\mathbb{N} \sqcup \{\infty\}$$