

# Type Systems and Programming

D. Renault

**ENSEIRB-Matmeca**

Apr. 9th 2025, v.1.5.1

<https://www.labri.fr/perso/renault/working/teaching/stp/stp.php>

# Introduction

What's a programming language?

```
int ackermann(int m, int n) {  
    if (!m) return n + 1;  
    if (!n) return ackermann(m-1,1);  
    return ackermann(m-1,  
                      ackermann(m,n-1));  
}
```

```
ackermann ← {  
    0 = 1 ⊃ ω : 1 + 2 ⊃ ω  
    0 = 2 ⊃ ω : ∇(¬1 + 1 ⊃ ω) 1  
    ∇(¬1 + 1 ⊃ ω), ∇(1 ⊃ ω), ¬1 + 2 ⊃ ω  
}
```

A complex and expressive tool for the representation of **computations**.

Focus on the problem of the **verification** of these computations.

What properties can one expect to be enforceable ?

- **Termination** properties : is it possible to be perfectly certain that a given program evaluates in a finite number of steps ?
- **Correctness** properties : is it possible to be perfectly certain that a program never ends up in an uncontrolled error state ?

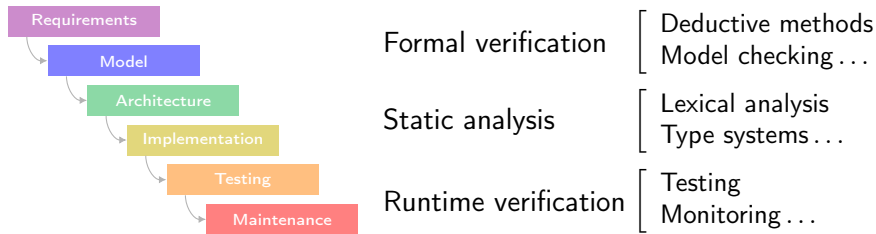
And more pragmatically, checking for the presence or absence of :

- null pointer exceptions, invalid file descriptors,
- indices out of array bounds, divisions by zero ...

# Introduction

How is it possible to enforce some of these properties ?

⇒ Different families of methods, spread along the development cycle.



⇒ Each family possesses different characteristics :

- Compile-time or Runtime
- Automatic or Assisted
- Decidable (complexity ?) or Semi-decidable

## Type systems

(informal description)

- a family of **tractable** methods,
- considering programs on a **syntactic** level,
- verifying some properties on their **behaviors**.

## General tactics

- Classify the expressions occurring inside a program into **types**,
- Verify that the combination of these **types** into the program respect a set of coherence rules.

Example :

`locomotive + flower`

Programming languages and type systems studied in this course :

- OCaml (4.14) [ocaml.org](https://ocaml.org)
- Haskell (ghc-9.4) [haskell.org/ghc](https://haskell.org/ghc)
- LiquidHaskell (0.9.4-git) [ucsd-progsys.github.io/liquidhaskell-blog](https://ucsd-progsys.github.io/liquidhaskell-blog)
- Scala (2.13) [scala-lang.org](https://scala-lang.org)

And their influence in mainstream languages :

- Java 8-21, C++ 14-23, C# 5-13 ...

# Some references

- Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.
- Bruce, K. B. *Foundations of Object-oriented Languages : Types and Semantics*. MIT Press, 2002.
- Hindley, J. R. *Basic simple type theory*. Cambridge University Press, 1997.
- Wadler, P. *Propositions as types*. Communications ACM, 2015.

## 1 Simple lambda-calculus

- Untyped
- Typed

## 2 Polymorphism

- Parametric
- Inclusion
- Adhoc

## 3 Subtyping

## 4 Proofs with types



- 1 Simple lambda-calculus
  - Propositional logic
  - Untyped lambda calculus
  - Simply typed lambda calculus
  - Type checking and inference
  - Curry-Howard correspondence

- 2 Polymorphism

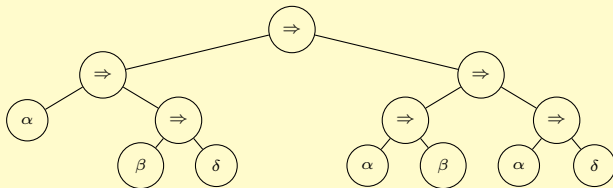
- 3 Subtyping

- 4 Proofs with types

## Definition (Minimal intuitionistic logic)

The **minimal intuitionistic logic** is the set of all formulae  $P, Q, \dots$  constructed from :

- an infinite set of atomic formulae denoted as variables  $\alpha, \beta, \dots$ ,
- if  $P, Q$  are two formulas, then  $P \Rightarrow Q$  is also a formula.



It is a simple fragment of the more general propositional logic.

## Definition (Sequent)

A **sequent** is an assertion  $\Gamma \vdash \alpha$ , where :

- $\Gamma$  is a possibly empty sequence of formulae called the **antecedents**,
- and  $\alpha$  is a formula called the **consequent**.

Writing  $\Gamma, P \vdash Q$  means that the antecedents are constituted of a list of formulae  $\Gamma$  along with a specific formula  $P$ .

## Definition (Derivation tree)

A **derivation tree** (or proof tree) is a tree whose nodes are syntactically coherent with a finite set of inference rules. In propositional logic, these rules are the following :

$$\frac{}{P \vdash P} [\text{AX}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} [\Rightarrow\text{I}] \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q} [\Rightarrow\text{E}]$$

Each inference rule possesses a name indicating its role, most of the time the introduction (I) or the elimination (E) of a logical operator.

## Frege's theorem

$$\left( R \Rightarrow (S \Rightarrow T) \right) \Rightarrow \left( (R \Rightarrow S) \Rightarrow (R \Rightarrow T) \right)$$

## Inference rules

$$\frac{}{P \vdash P} [\text{AX}]$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} [\Rightarrow\text{I}]$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q} [\Rightarrow\text{E}]$$

## Proof as a derivation tree

$$\frac{\frac{\frac{}{P \vdash P} [\text{AX}]}{P \vdash P} \quad \frac{\frac{\frac{}{R \vdash R} [\text{AX}]}{R \vdash R} \quad \frac{\frac{}{S \vdash S} [\text{AX}]}{S \vdash S}}{R \vdash S \Rightarrow T} [\Rightarrow\text{I}]}{R \vdash (S \Rightarrow T)} [\Rightarrow\text{I}]}{\vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))} [\Rightarrow\text{I}]$$

## Frege's theorem

$$\left( R \Rightarrow (S \Rightarrow T) \right) \Rightarrow \left( (R \Rightarrow S) \Rightarrow (R \Rightarrow T) \right)$$

## Inference rules

$$\frac{}{P \vdash P} [\text{AX}] \quad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} [\Rightarrow \text{I}] \quad \frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q} [\Rightarrow \text{E}]$$

## Proof as a derivation tree

$$\frac{(R \Rightarrow (S \Rightarrow T)) \vdash (R \Rightarrow S) \Rightarrow (R \Rightarrow T)}{\vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))}$$

## Frege's theorem

$$\left( R \Rightarrow (S \Rightarrow T) \right) \Rightarrow \left( (R \Rightarrow S) \Rightarrow (R \Rightarrow T) \right)$$

## Inference rules

$$\frac{}{P \vdash P} [\text{AX}]$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} [\Rightarrow\text{I}]$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q} [\Rightarrow\text{E}]$$

## Proof as a derivation tree

$$\frac{\frac{\frac{\frac{}{(R \Rightarrow (S \Rightarrow T))}, (R \Rightarrow S) \vdash (R \Rightarrow T)}{(R \Rightarrow (S \Rightarrow T)) \vdash (R \Rightarrow S) \Rightarrow (R \Rightarrow T)}}{\vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))}}{\vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))}}$$

## Frege's theorem

$$\left( R \Rightarrow (S \Rightarrow T) \right) \Rightarrow \left( (R \Rightarrow S) \Rightarrow (R \Rightarrow T) \right)$$

## Inference rules

$$\frac{}{P \vdash P} [\text{AX}] \quad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} [\Rightarrow\text{I}] \quad \frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q} [\Rightarrow\text{E}]$$

## Proof as a derivation tree

$$\frac{\frac{\frac{\Gamma ::= \{(R \Rightarrow (S \Rightarrow T)), (R \Rightarrow S), R\} \vdash T}{(R \Rightarrow (S \Rightarrow T)), (R \Rightarrow S) \vdash (R \Rightarrow T)}}{(R \Rightarrow (S \Rightarrow T)) \vdash (R \Rightarrow S) \Rightarrow (R \Rightarrow T)}}{\vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))}$$



## Frege's theorem

$$\left( R \Rightarrow (S \Rightarrow T) \right) \Rightarrow \left( (R \Rightarrow S) \Rightarrow (R \Rightarrow T) \right)$$

## Inference rules

$$\frac{}{P \vdash P} [\text{AX}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} [\Rightarrow \text{I}] \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q} [\Rightarrow \text{E}]$$

## Proof as a derivation tree

$$\frac{\frac{\frac{\frac{\frac{\frac{\Gamma \vdash S \Rightarrow T}{\Gamma ::= \{(R \Rightarrow (S \Rightarrow T)), (R \Rightarrow S), R\} \vdash T}{(R \Rightarrow (S \Rightarrow T)), (R \Rightarrow S) \vdash (R \Rightarrow T)}}{(R \Rightarrow (S \Rightarrow T)) \vdash (R \Rightarrow S) \Rightarrow (R \Rightarrow T)}}{\vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))}}{\Gamma \vdash S} \quad \frac{}{\Gamma \vdash S}$$

## Frege's theorem

$$\left( R \Rightarrow (S \Rightarrow T) \right) \Rightarrow \left( (R \Rightarrow S) \Rightarrow (R \Rightarrow T) \right)$$

## Inference rules

$$\frac{}{P \vdash P} [\text{AX}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} [\Rightarrow \text{I}] \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q} [\Rightarrow \text{E}]$$

## Proof as a derivation tree

$$\frac{\frac{\frac{\frac{\frac{\frac{\frac{\Gamma \vdash S \Rightarrow T}{\Gamma :: \{(R \Rightarrow (S \Rightarrow T)), (R \Rightarrow S), R\} \vdash T}{(R \Rightarrow (S \Rightarrow T)), (R \Rightarrow S) \vdash (R \Rightarrow T)}{(R \Rightarrow (S \Rightarrow T)) \vdash (R \Rightarrow S) \Rightarrow (R \Rightarrow T)}}{\vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))}}{\frac{\Gamma \vdash R \quad \Gamma \vdash R \Rightarrow S}{\Gamma \vdash S}}}$$

## Frege's theorem

$$\left( R \Rightarrow (S \Rightarrow T) \right) \Rightarrow \left( (R \Rightarrow S) \Rightarrow (R \Rightarrow T) \right)$$

## Inference rules

$$\frac{}{P \vdash P} [\text{AX}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} [\Rightarrow \text{I}] \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q} [\Rightarrow \text{E}]$$

## Proof as a derivation tree

$$\frac{\frac{\frac{\Gamma \vdash R}{\Gamma \vdash S \Rightarrow T} \quad \frac{\Gamma \vdash R \Rightarrow (S \Rightarrow T)}{\Gamma \vdash S}}{\Gamma \vdash S \Rightarrow T} \quad \frac{\frac{\Gamma \vdash R}{\Gamma \vdash S} \quad \frac{\Gamma \vdash R \Rightarrow S}{\Gamma \vdash S}}{\Gamma \vdash S} \quad \frac{\Gamma \vdash S \Rightarrow T \quad \Gamma \vdash S}{\Gamma \vdash T} \quad \frac{\Gamma \vdash T}{\Gamma \vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))} \quad \frac{\Gamma \vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))}{\vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))}$$

## Frege's theorem

$$\left( R \Rightarrow (S \Rightarrow T) \right) \Rightarrow \left( (R \Rightarrow S) \Rightarrow (R \Rightarrow T) \right)$$

## Inference rules

$$\frac{}{P \vdash P} [\text{AX}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} [\Rightarrow \text{I}] \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q} [\Rightarrow \text{E}]$$

## Proof as a derivation tree

$$\frac{\frac{\frac{\Gamma \vdash R}{\Gamma \vdash S \Rightarrow T} \quad \frac{\Gamma \vdash R \Rightarrow (S \Rightarrow T)}{\Gamma \vdash S \Rightarrow T}}{\Gamma \vdash S \Rightarrow T} \quad \frac{\frac{\Gamma \vdash R}{\Gamma \vdash S} \quad \frac{\Gamma \vdash R \Rightarrow S}{\Gamma \vdash S}}{\Gamma \vdash S} \quad \frac{\Gamma \vdash S \Rightarrow T \quad \Gamma \vdash S}{\Gamma \vdash (R \Rightarrow (S \Rightarrow T)), (R \Rightarrow S), R \vdash T} \quad \frac{\Gamma \vdash (R \Rightarrow (S \Rightarrow T)), (R \Rightarrow S), R \vdash T}{(R \Rightarrow (S \Rightarrow T)), (R \Rightarrow S) \vdash (R \Rightarrow T)} \quad \frac{(R \Rightarrow (S \Rightarrow T)), (R \Rightarrow S) \vdash (R \Rightarrow T)}{(R \Rightarrow (S \Rightarrow T)) \vdash (R \Rightarrow S) \Rightarrow (R \Rightarrow T)} \quad \frac{(R \Rightarrow (S \Rightarrow T)) \vdash (R \Rightarrow S) \Rightarrow (R \Rightarrow T)}{\vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))}$$



# Summary on propositional logic

The model of propositional logic offers :

- a **language** describing a family of objects **inductively**,
- and a system for defining a subset of this family respecting **local rules**.

The difficulty lies in constructing a kind of **proof** (here a derivation tree) for assessing the validity of a proposition.

In the following, we construct an equivalent model for a programming language : the **untyped  $\lambda$ -calculus**.

So let's start again ...

## Definition (Untyped $\lambda$ -calculus)

The untyped  $\lambda$ -calculus is the set of expressions  $t, u, \dots$  constructed from :

- **[Variable]** an infinite set of abstract variables  $x, y, \dots$ ,
- **[Abstraction]** if  $t$  is an expression and  $x$  is a variable, then  $\lambda x.t$  is an expression,
- **[Application]** if  $t, u$  are two expressions, then  $(t\ u)$  is an expression.

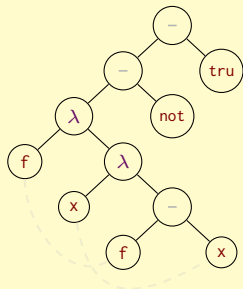
### Example :

$((\lambda f.\lambda x.(f\ x)\text{-not})\text{-true})$

Python : `(lambda f: lambda x: f(x))(__not__)(True)`

Scheme : `((lambda (f) (lambda (x) (f x))) not) #t)`

OCaml : `(fun f → fun x → f(x))(not)(true)`



## Definition (Free / Bound variables)

A variable  $x$  in a  $\lambda$ -expression  $u$  is said to be **bound** iff it appears as a descendant of an abstraction node over the same variable  $x$ . Otherwise, it is said to be **free**.

$FV(u)$ , the free variables of  $u$  :

- $FV(x) = \{x\}$
- $FV(u.v) = FV(u) \cup FV(v)$
- $FV(\lambda x.u) = FV(u) \setminus \{x\}$

$BV(u)$ , the bound variables of  $u$  :

- $BV(x) = \{\}$
- $BV(u.v) = BV(u) \cup BV(v)$
- $BV(\lambda x.u) = BV(u) \cup \{x\}$

## Examples

- $FV((\lambda f.\lambda x.(f\ x)\_not)\_true) = \{not, true\}$
- $BV((\lambda f.\lambda x.(f\ x)\_not)\_true) = \{f, x\}$

## Definition ( $\alpha$ -conversion)

Let  $t ::= \lambda x.u$  be an expression and  $y$  a variable. An  **$\alpha$ -conversion** of  $t$  is an expression  $\lambda y.v$  where  $v$  is a copy of  $u$  where every free variable  $x$  in  $u$  has been replaced by  $y$ .

... where  $r_{x \rightarrow y}(t)$  is defined as :

$\alpha\text{cnv}_{x \rightarrow y}(t)$  is defined as :

- $\alpha\text{cnv}_{x \rightarrow y}(\lambda x.u) = \lambda y.r_{x \rightarrow y}(v)$
- $\alpha\text{cnv}_{x \rightarrow y}(t) = t$  otherwise
- $r_{x \rightarrow y}(\lambda z.w) = \lambda z.w$  if  $z = x$ ,  
 $= \lambda z.r_{x \rightarrow y}(w)$  otherwise
- $r_{x \rightarrow y}(v.w) = (r_{x \rightarrow y}(v).r_{x \rightarrow y}(w))$

## Examples

- $\alpha\text{cnv}_{x \rightarrow y}(\lambda x.x) = \lambda y.y$
- $\alpha\text{cnv}_{x \rightarrow y}(\lambda x.((\lambda x.x)_x)) = \lambda y.((\lambda x.x)_y)$





## Definition (Substitution)

Let  $t, u$  be  $\lambda$ -expressions and  $x$  a variable. To **substitute**  $x$  by  $u$  into  $t$ , noted  $[x \mapsto u]t$ , consists in replacing every free occurrence of  $x$  in  $t$  by a copy of  $u$ .

$[x \mapsto u]t$  is defined as :

- $[x \mapsto u] z = u$  if  $z = x$ ,  $z$  otherwise
- $[x \mapsto u](v . w) = ([x \mapsto u]v . [x \mapsto u]w)$
- $[x \mapsto u] \lambda z . w = \lambda z . [x \mapsto u]w$  if  $z \neq x$  and  $z \notin FV(u)$ ,  
 $\lambda z . w$  otherwise.

## Example

$$[x \mapsto \text{biiip}] \left( (\lambda z . (x . z)) . y \right) = (\lambda z . (\text{biiip} . z)) . y$$

## Definition ( $\beta$ -reduction)

A **redex** in a  $\lambda$ -expression  $t$  is a sub-expression of the form  $((\lambda x.v) \_ w)$ . Applying a  **$\beta$ -reduction** step from  $t$  to  $u$ , noted  $t \rightarrow_{\beta} u$ , consists in finding a redex sub-expression  $((\lambda x.v) \_ w)$  inside  $t$  and replacing it by  $[x \mapsto w]v$ .

$t \rightarrow_{\beta} u$  is defined as :

- $(\lambda x.v) \_ w \rightarrow_{\beta} [x \mapsto w]v$
  - if  $u \rightarrow_{\beta} v$  then
    - Function reduction :  $(u \_ w) \rightarrow_{\beta} (v \_ w)$
    - Parameter reduction :  $(w \_ u) \rightarrow_{\beta} (w \_ v)$
    - Body reduction :  $\lambda x.u \rightarrow_{\beta} \lambda x.v$
- $\left. \begin{array}{l} \text{Weak red.} \\ \text{Strong red.} \end{array} \right\}$

## Example

$$\left( \lambda z. (\lambda x. x+1) \_ (z+2) \right) \_ (3+4) \rightarrow_{\beta} \dots \rightarrow_{\beta} ((3+4)+2)+1$$

## Definition ( $\beta$ -reduction)

A **redex** in a  $\lambda$ -expression  $t$  is a sub-expression of the form  $((\lambda x.v)_w)$ . Applying a  **$\beta$ -reduction** step from  $t$  to  $u$ , noted  $t \rightarrow_\beta u$ , consists in finding a redex sub-expression  $((\lambda x.v)_w)$  inside  $t$  and replacing it by  $[x \mapsto w]v$ .

$t \rightarrow_\beta u$  is defined as :

- $(\lambda x.v)_w \rightarrow_\beta [x \mapsto w]v$
  - if  $u \rightarrow_\beta v$  then
    - Function reduction :  $(u_w) \rightarrow_\beta (v_w)$
    - Parameter reduction :  $(w_u) \rightarrow_\beta (w_v)$
    - Body reduction :  $\lambda x.u \rightarrow_\beta \lambda x.v$
- Weak red. }  
Strong red. }

- An expression to which no  $\beta$ -reduction step can be applied is said to be in **normal form**.
- The evaluation of a  $\lambda$ -expression consists in applying  $\beta$ -reductions as long as it is possible.

# Properties of the $\lambda$ -calculus

The  $\lambda$ -calculus endowed with the  $\beta$ -reduction relation is a Turing-complete computational model.

- **Church-Rosser theorem** : the  $\beta$ -reduction relation is **confluent**.
- There exist  $\lambda$ -expressions for which the evaluation is **infinite**.

$$nt ::= (\lambda x. (x \_ x)) \_ (\lambda x. (x \_ x)) \qquad nt \rightarrow_{\beta} nt$$

- **Church undecidability theorem** : the problem of determining whether the evaluation of a  $\lambda$ -expression is finite or not is **undecidable**.

Undecidability is at the heart of dealing with programming languages.

Syntax	Evaluation rules
$t ::=$	
$x$ <i>expressions</i>	
$\lambda x.t$ <i>variable</i>	
$(t_1 t_2)$ <i>abstraction</i>	
$(t_1 t_2)$ <i>application</i>	
$v ::=$	
$\lambda x.t$ <i>values</i>	
$\lambda x.t$ <i>abstraction value</i>	

- **Values** are particular expressions that need no more evaluation.
- In this model, the values are **exactly** the expressions in normal form.

# Church encodings : booleans

Let us encode the classical boolean values as the following expressions :

$$\begin{aligned}\text{true} &::= \lambda x. \lambda y. x \\ \text{false} &::= \lambda x. \lambda y. y\end{aligned}$$

(these are the classical projection functions)

The booleans can now be manipulated with the following expressions :

$$\begin{aligned}\text{if} &::= \lambda b. \lambda t. \lambda e. ((b\ t)\ e) \\ \text{or} &::= \lambda x. \lambda y. (((\text{if}\ x)\ \text{true})\ y) \\ \text{and} &::= \lambda x. \lambda y. (((\text{if}\ x)\ y)\ \text{false}) \\ \text{not} &::= \lambda x. (((\text{if}\ x)\ \text{false})\ \text{true})\end{aligned}$$

## Example

$$\text{or true false} \rightarrow_{\beta}$$

# Church encodings : booleans

Let us encode the classical boolean values as the following expressions :

$$\begin{aligned}\text{true} &::= \lambda x. \lambda y. x \\ \text{false} &::= \lambda x. \lambda y. y\end{aligned}$$

(these are the classical projection functions)

The booleans can now be manipulated with the following expressions :

$$\begin{aligned}\text{if} &::= \lambda b. \lambda t. \lambda e. ((b\ t)\ e) \\ \text{or} &::= \lambda x. \lambda y. (((\text{if}\ x)\ \text{true})\ y) \\ \text{and} &::= \lambda x. \lambda y. (((\text{if}\ x)\ y)\ \text{false}) \\ \text{not} &::= \lambda x. (((\text{if}\ x)\ \text{false})\ \text{true})\end{aligned}$$

## Example

$$\text{or true false} \rightarrow_{\beta} \text{if true true false} \rightarrow_{\beta}$$



# Church encodings : booleans

Let us encode the classical boolean values as the following expressions :

$\text{true} ::= \lambda x. \lambda y. x$   
 $\text{false} ::= \lambda x. \lambda y. y$

(these are the classical projection functions)

The booleans can now be manipulated with the following expressions :

$\text{if} ::= \lambda b. \lambda t. \lambda e. ((b\ t)\ e)$   
 $\text{or} ::= \lambda x. \lambda y. (((\text{if}\ x)\ \text{true})\ y)$   
 $\text{and} ::= \lambda x. \lambda y. (((\text{if}\ x)\ y)\ \text{false})$   
 $\text{not} ::= \lambda x. (((\text{if}\ x)\ \text{false})\ \text{true})$

## Example

$\text{or}\ \text{true}\ \text{false} \rightarrow_{\beta} \text{if}\ \text{true}\ \text{true}\ \text{false} \rightarrow_{\beta} \text{true}\ \text{true}\ \text{false} \rightarrow_{\beta}$

# Church encodings : booleans

Let us encode the classical boolean values as the following expressions :

$$\begin{aligned}\text{true} &::= \lambda x. \lambda y. x \\ \text{false} &::= \lambda x. \lambda y. y\end{aligned}$$

(these are the classical projection functions)

The booleans can now be manipulated with the following expressions :

$$\begin{aligned}\text{if} &::= \lambda b. \lambda t. \lambda e. ((b\ t)\ e) \\ \text{or} &::= \lambda x. \lambda y. (((\text{if}\ x)\ \text{true})\ y) \\ \text{and} &::= \lambda x. \lambda y. (((\text{if}\ x)\ y)\ \text{false}) \\ \text{not} &::= \lambda x. (((\text{if}\ x)\ \text{false})\ \text{true})\end{aligned}$$

## Example

$$\text{or true false} \rightarrow_{\beta} \text{if true true false} \rightarrow_{\beta} \text{true true false} \rightarrow_{\beta} \text{true}$$

# Church encodings : naturals

And the following expressions encode the natural numbers :

$\text{zero} ::= \lambda f. \lambda x. x$   $f$  applied zero times  
 $\text{succ} ::= \lambda i. \lambda f. \lambda x. (f \_ ((i \_ f) \_ x))$   $f$  applied once to the result of  $(i \_ f)$

With the addition and multiplication functions defined as follows :

$\text{plus} ::= \lambda i. \lambda j. ((i \_ \text{succ}) \_ ((j \_ \text{succ}) \_ \text{zero}))$  apply  $\text{succ}$ , first  $j$  times then  $i$  times to  $\text{zero}$   
 $\text{mult} ::= \lambda i. \lambda j. ((j \_ (\text{plus} \_ i)) \_ \text{zero})$  apply  $(\text{plus} \_ i)$ ,  $j$  times to  $\text{zero}$

# Example : derivation tree of a $\beta$ -reduction

In the  $\lambda$ -calculus extended with the Church boolean values :

$$\begin{array}{c}
 \frac{\text{(if\_true)} \rightarrow_{\beta} \lambda t. \lambda e. ((\text{true\_t})\_e)}{\text{((if\_true)\_false)} \rightarrow_{\beta} (\lambda t. \lambda e. ((\text{true\_false})\_e)\_false) \rightarrow_{\beta} \lambda e. ((\text{true\_false})\_e)} \\
 \text{(not\_true)} \rightarrow_{\beta} \frac{\text{(((if\_true)\_false)\_true)} \rightarrow_{\beta} (\lambda e. ((\text{true\_false})\_e)\_true)}{\dots \rightarrow_{\beta} \text{((true\_false)\_true)} \rightarrow_{\beta} (\lambda y. \text{false\_true}) \rightarrow_{\beta} \text{false}}
 \end{array}$$

The untyped  $\lambda$ -calculus is **everything but practical** :

- Its evaluation rule is remarkably simple.
- But the encodings are multiple and possibly overlapping.

## Improvement idea

Extend the language with new expressions : **true**, **succ**, **zero** ...

Possible advantages : higher level of abstraction, custom constructs and values in the language, specific evaluation rules ...

But it becomes necessary to deal with expressions such as **succ true**.

$$(\text{succ\_true}) \rightarrow_{\beta} \underbrace{(\lambda i f x. (f\_((i\_f)\_x)))}_{\text{succ}} \underbrace{(\lambda u v. u)}_{\text{true}} \rightarrow_{\beta} \underbrace{\lambda f x. (f\_f)}_{??}$$

Let's do it anyway ...

# Extension of the $\lambda$ -calculus : booleans & naturals

## Syntax

$t ::= \dots$  *expressions*  
true, false *booleans*  
zero, succ t *naturals*  
if t then t else t *if-then-else*  
iszero t *zero-equality*

$v ::= \dots$  *values*  
true, false *boolean value*  
nv *numeric value*

$nv ::=$  *numeric values*  
zero *zero value*  
succ nv *successor value*

## Evaluation rules

$$\frac{t_1 \rightarrow_{\beta} t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow_{\beta} \text{if } t'_1 \text{ then } t_2 \text{ else } t_3}$$

if true then  $t_2$  else  $t_3 \rightarrow_{\beta} t_2$   
if false then  $t_2$  else  $t_3 \rightarrow_{\beta} t_3$

$$\frac{t \rightarrow_{\beta} t'}{\text{iszero } t \rightarrow_{\beta} \text{iszero } t'}$$

iszero zero  $\rightarrow_{\beta}$  true  
iszero (succ t)  $\rightarrow_{\beta}$  false

# Example : derivation tree of a $\beta$ -reduction

Examples of evaluation in the  $\lambda$ -calculus with booleans and integers :

- $$\frac{\text{iszero (succ zero)} \quad \rightarrow_{\beta} \quad \text{false}}{\text{if (iszero (succ zero)) then zero else (succ zero)} \rightarrow_{\beta} \text{if false then zero else (succ zero)} \rightarrow_{\beta} \text{succ zero}}$$
- $$\frac{\text{if false then zero else false} \quad \rightarrow_{\beta} \quad \text{false}}{\text{succ (if false then zero else false)} \rightarrow_{\beta} \text{succ false} \rightarrow_{\beta} ??}$$

## Problem

This new language contains **stuck** expressions, that cannot be evaluated further but are still not values, e.g `succ true` or `if zero then true else false`.

- These expressions are the sign of an indecision in the evaluation relation.
- They occur because most of the interesting functions are **partial**.

# Summary on the untyped $\lambda$ -calculus

Starting from now, we consider that the booleans and naturals are part of the definition of the  $\lambda$ -calculus.

- The obtained language is close to a classical programming language without side effects.
- There exist **stuck** expressions that are neither values nor in normal form.
- Stuck expressions are mostly **unavoidable** when extending the language.

In the following we shall endow this language with **types** that allow to determine whether an expression is stuck or not without its full evaluation.



# From untyped to typed

Recall our general approach :

## General tactics

- Classify the expressions occurring inside a program into **types**,
- Verify that the composition of these types into the program respects a set of **coherence rules**.

In order to do this, we shall define a set of types and rules such that :

- a **type** acts as an **approximation of the evaluation of an expression** ;
- a **rule** is associated to a syntactic construct of the language and expresses **how this construct evaluates** with regard to types.

These types and rules shall define a **type system**.

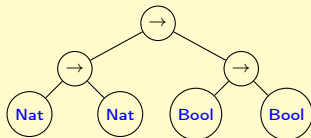
## Definition (Types)

The set of **types**, noted **Typ**, is defined as :

- **Type variable** : an infinite set of abstract type variables  $T, U, \dots$
- **Function type** : if  $T$  and  $U$  are types, then  $T \rightarrow U$  is also a type.
- In our setting, we add two constant types : **Nat** and **Bool**.
- A type is **concrete** iff it contains only constant types as sub-expressions.

## Example

$(\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Bool} \rightarrow \text{Bool})$



## Example : the if-then-else construct

Consider an expression  $t_{if} ::= \text{if } t_1 \text{ then } t_2 \text{ else } t_3$  that should be checked.

An **inference rule** for the **if** construct should :

- assume a series of properties on the types of  $t_1$ ,  $t_2$  and  $t_3$ ,
- and deduce a property on the type of  $t_{if}$ .

Key : a type approximates the **result of the evaluation** of an expression.

$$\frac{\text{Assumption on } t_1 \quad \text{Assumption on } t_2 \quad \text{Assumption on } t_3}{\text{Assumption on } \text{if } t_1 \text{ then } t_2 \text{ else } t_3}$$

## Example : the if-then-else construct

Consider an expression  $t_{if} ::= \text{if } t_1 \text{ then } t_2 \text{ else } t_3$  that should be checked.

An **inference rule** for the **if** construct should :

- assume a series of properties on the types of  $t_1$ ,  $t_2$  and  $t_3$ ,
- and deduce a property on the type of  $t_{if}$ .

Key : a type approximates the **result of the evaluation** of an expression.

$$\frac{\text{if } t_1 \text{ has type Bool} \quad t_2 \text{ has type } T \quad t_3 \text{ has the same type as } t_2}{\text{then if } t_1 \text{ then } t_2 \text{ else } t_3 \text{ has type } T}$$

- An expression  $t$  is said to **have type**  $T \in \mathbf{Typ}$ , noted  $t : T$ .  
This yields a **typing**, an association between an expression and a type.
- An **environment**  $\Gamma$  is a possibly empty sequence of typings.

### Definition (Typing deduction)

To **deduce a typing** from  $\Gamma$ , noted  $\Gamma \vdash t : T$ , consists in building a derivation tree using  $\Gamma$  as a set of axioms and a finite set of typing rules, whose root asserts that  $t : T$ .

- An expression  $t$  is said to be **typable** if it is possible to deduce a typing  $T$  for  $t$  starting from the empty environment.
- As a consequence, the expression  $t : T$  is said to be **(well)-typed**.

## Example : the if-then-else construct

For the **if-then-else** construct  $t_{if} ::= \text{if } t_1 \text{ then } t_2 \text{ else } t_3$ .

Suppose that in an environment  $\Gamma$  :

- one can prove that  $t_1 : \text{Bool}$ ,
- one can prove that  $t_2 : T$  for a particular  $T$ ,
- one can prove that  $t_3 : T$ ,

Then we deduce that  $t_{if} : T$ .

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

## Example : application and abstraction

For the **application** construct  $t_{app} ::= (t_1 - t_2)$  in an environment  $\Gamma$ .

$$\frac{\Gamma \vdash t_1 \quad \Gamma \vdash t_2}{\Gamma \vdash (t_1 - t_2)}$$

## Example : application and abstraction

For the **application** construct  $t_{app} ::= (t_1 \cdot t_2)$  in an environment  $\Gamma$ .

$$\frac{\Gamma \vdash t_1 : T \rightarrow U \quad \Gamma \vdash t_2 : T}{\Gamma \vdash (t_1 \cdot t_2) : U}$$



## Example : application and abstraction

For the **application** construct  $t_{app} ::= (t_1 \_ t_2)$  in an environment  $\Gamma$ .

$$\frac{\Gamma \vdash t_1 : T \rightarrow U \quad \Gamma \vdash t_2 : T}{\Gamma \vdash (t_1 \_ t_2) : U}$$

For the **abstraction** construct  $t_{abs} ::= \lambda x. t_1$  in an environment  $\Gamma$ .

## Example : application and abstraction

For the **application** construct  $t_{app} ::= (t_1 \cdot t_2)$  in an environment  $\Gamma$ .

$$\frac{\Gamma \vdash t_1 : T \rightarrow U \quad \Gamma \vdash t_2 : T}{\Gamma \vdash (t_1 \cdot t_2) : U}$$

For the **abstraction** construct  $t_{abs} ::= \lambda x. t_1$  in an environment  $\Gamma$ .

$$\frac{}{\Gamma \vdash \lambda x. t_1}$$

## Example : application and abstraction

For the **application** construct  $t_{app} ::= (t_1 \text{ } t_2)$  in an environment  $\Gamma$ .

$$\frac{\Gamma \vdash t_1 : T \rightarrow U \quad \Gamma \vdash t_2 : T}{\Gamma \vdash (t_1 \text{ } t_2) : U}$$

For the **abstraction** construct  $t_{abs} ::= \lambda x. t_1$  in an environment  $\Gamma$ .

$$\frac{\Gamma, x \vdash t_1}{\Gamma \vdash \lambda x. t_1}$$

## Example : application and abstraction

For the **application** construct  $t_{app} ::= (t_1 \text{ } t_2)$  in an environment  $\Gamma$ .

$$\frac{\Gamma \vdash t_1 : T \rightarrow U \quad \Gamma \vdash t_2 : T}{\Gamma \vdash (t_1 \text{ } t_2) : U}$$

For the **abstraction** construct  $t_{abs} ::= \lambda x. t_1$  in an environment  $\Gamma$ .

$$\frac{\Gamma, x : T \vdash t_1 : U}{\Gamma \vdash \lambda x. t_1 : T \rightarrow U}$$

# What about the typed abstraction ?

Consider the **typed abstraction** construct :  $t_{\text{tabs}} ::= \lambda x : T. t_1$

With nearly the same typing rule :

$$\frac{\Gamma, x : T \vdash t_1 : U}{\Gamma \vdash \lambda x : T. t_1 : T \rightarrow U}$$

Annotating the code with types or not offers different perspectives :

- **Explicit** types : simpler (or even just decidable) verification.
- **Implicit** types : no-hassle programming, principal types.

```
vector<int> list;  
for (auto it = list.begin(); it != list.end(); it++)  
    cout << *it << endl;           // in place of 'vector<int>::iterator'
```

# Example : derivation tree of a typing

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

$\emptyset \vdash \lambda f. \lambda x. (f \_ (f \_ x)) :$

# Example : derivation tree of a typing

$$\frac{\frac{\frac{}{\{f : \quad\} \vdash \lambda x. (f \_ (f \_ x)) :}}{\quad}}{\quad}}{\emptyset \vdash \lambda f. \lambda x. (f \_ (f \_ x)) :}$$

## Example : derivation tree of a typing

$$\frac{\frac{\frac{}{\Gamma ::= \{f : \quad, x : \quad\} \vdash (f\_ (f\_ x)) :}}{\{f : \quad\} \vdash \lambda x. (f\_ (f\_ x)) :}}{\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_ x)) :}$$



## Example : derivation tree of a typing

$$\frac{\frac{\Gamma \vdash f : \quad \quad \quad \Gamma \vdash (f\_x) :}{\Gamma ::= \{f : \quad \quad \quad , x : \quad \quad \} \vdash (f\_ (f\_x)) :}}{\{f : \quad \quad \quad \} \vdash \lambda x. (f\_ (f\_x)) :}}{\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) :}$$

## Example : derivation tree of a typing

$$\frac{\frac{f : \quad \in \Gamma}{\Gamma \vdash f : \quad} \quad \frac{\quad}{\Gamma \vdash (f\_x) : \quad}}{\Gamma ::= \{f : \quad, x : \quad\} \vdash (f\_ (f\_x)) : \quad}}{\{f : \quad\} \vdash \lambda x. (f\_ (f\_x)) : \quad}}{\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : \quad}$$

## Example : derivation tree of a typing

$$\frac{\frac{f : \quad \in \Gamma}{\Gamma \vdash f : \quad} \quad \frac{\Gamma \vdash f : \quad}{\Gamma \vdash (f\_x) : \quad} \quad \frac{\Gamma \vdash x : \quad}{\Gamma \vdash (f\_x) : \quad}}{\Gamma ::= \{f : \quad, x : \quad\} \vdash (f\_x) : \quad} \\ \frac{\{f : \quad\} \vdash \lambda x. (f\_x) : \quad}{\emptyset \vdash \lambda f. \lambda x. (f\_x) : \quad}$$

## Example : derivation tree of a typing

$$\frac{\frac{f : \quad \in \Gamma}{\Gamma \vdash f :} \quad \frac{\frac{f : \quad \in \Gamma}{\Gamma \vdash f :} \quad \frac{x : \quad \in \Gamma}{\Gamma \vdash x :}}{\Gamma \vdash (f\_x) :} \\ \frac{\Gamma ::= \{f : \quad , x : \quad \} \vdash (f\_ (f\_x)) :}{\{f : \quad \} \vdash \lambda x. (f\_ (f\_x)) :} \\ \frac{}{\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) :}$$

## Example : derivation tree of a typing

$$\frac{\frac{f : \quad \in \Gamma}{\Gamma \vdash f : \quad} \quad \frac{\frac{f : \quad \in \Gamma}{\Gamma \vdash f : \quad} \quad \frac{x : \text{Nat} \in \Gamma}{\Gamma \vdash x : \text{Nat}}}{\Gamma \vdash (f\_x) : \text{Nat}} \\ \frac{\Gamma ::= \{f : \quad, x : \text{Nat}\} \vdash (f\_ (f\_x)) : \text{Nat}}{\{f : \quad\} \vdash \lambda x. (f\_ (f\_x)) : \quad} \\ \hline \emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) :$$

## Example : derivation tree of a typing

$$\frac{\frac{f : \text{Nat} \rightarrow \text{Nat} \in \Gamma}{\Gamma \vdash f : \text{Nat} \rightarrow \text{Nat}} \quad \frac{\frac{f : \text{Nat} \rightarrow \text{Nat} \in \Gamma}{\Gamma \vdash f : \text{Nat} \rightarrow \text{Nat}} \quad \frac{x : \text{Nat} \in \Gamma}{\Gamma \vdash x : \text{Nat}}}{\Gamma \vdash (f\_x) : \text{Nat}} \\ \frac{\Gamma ::= \{f : \text{Nat} \rightarrow \text{Nat}, x : \text{Nat}\} \vdash (f\_ (f\_x)) : \text{Nat}}{\{f : \text{Nat} \rightarrow \text{Nat}\} \vdash \lambda x. (f\_ (f\_x)) : \text{Nat} \rightarrow \text{Nat}} \\ \frac{}{\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : }$$

## Example : derivation tree of a typing

$$\frac{\frac{f : \text{Nat} \rightarrow \text{Nat} \in \Gamma}{\Gamma \vdash f : \text{Nat} \rightarrow \text{Nat}} \quad \frac{\frac{f : \text{Nat} \rightarrow \text{Nat} \in \Gamma}{\Gamma \vdash f : \text{Nat} \rightarrow \text{Nat}} \quad \frac{x : \text{Nat} \in \Gamma}{\Gamma \vdash x : \text{Nat}}}{\Gamma \vdash (f\_x) : \text{Nat}} \\ \frac{\Gamma ::= \{f : \text{Nat} \rightarrow \text{Nat}, x : \text{Nat}\} \vdash (f\_ (f\_x)) : \text{Nat}}{\{f : \text{Nat} \rightarrow \text{Nat}\} \vdash \lambda x. (f\_ (f\_x)) : \text{Nat} \rightarrow \text{Nat}} \\ \frac{}{\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat} \rightarrow \text{Nat}}$$

## Example : derivation tree of a typing

$$\frac{\frac{f : \text{Nat} \rightarrow \text{Nat} \in \Gamma}{\Gamma \vdash f : \text{Nat} \rightarrow \text{Nat}} \quad \frac{\frac{f : \text{Nat} \rightarrow \text{Nat} \in \Gamma}{\Gamma \vdash f : \text{Nat} \rightarrow \text{Nat}} \quad \frac{x : \text{Nat} \in \Gamma}{\Gamma \vdash x : \text{Nat}}}{\Gamma \vdash (f\_x) : \text{Nat}} \\ \frac{\Gamma ::= \{f : \text{Nat} \rightarrow \text{Nat}, x : \text{Nat}\} \vdash (f\_ (f\_x)) : \text{Nat}}{\{f : \text{Nat} \rightarrow \text{Nat}\} \vdash \lambda x. (f\_ (f\_x)) : \text{Nat} \rightarrow \text{Nat}} \\ \frac{}{\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat} \rightarrow \text{Nat}}$$





The **simply-typed  $\lambda$ -calculus** or  $\lambda_{\rightarrow}$  is defined as the set of the typable  $\lambda$ -expressions in the **Typ** family of types with the following typing rules :

$$\frac{t : T \in \Gamma}{\Gamma \vdash t : T} [\text{VAR}]$$
$$\frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x. u : T \rightarrow U} [\text{ABS}]$$
$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash f : T \rightarrow U}{\Gamma \vdash (f \_ x) : U} [\text{APP}]$$

The **simply-typed  $\lambda$ -calculus** or  $\lambda_{\rightarrow}$  is defined as the set of the typable  $\lambda$ -expressions in the **Typ** family of types with the following typing rules :

$$\begin{array}{c}
 \frac{t : T \in \Gamma}{\Gamma \vdash t : T} [\text{VAR}] \\
 \frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x. u : T \rightarrow U} [\text{ABS}] \\
 \frac{\Gamma \vdash x : T \quad \Gamma \vdash f : T \rightarrow U}{\Gamma \vdash (f \ x) : U} [\text{APP}]
 \end{array}$$

Comparison with the rules in propositional logic :

$$\frac{}{P \vdash P} [\text{AX}] \quad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} [\Rightarrow I] \quad \frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q} [\Rightarrow E]$$

# Typing rules for booleans and naturals

$$\Gamma \vdash \text{true}$$
$$\Gamma \vdash \text{false}$$
$$\frac{\Gamma \vdash t_1 \quad \Gamma \vdash t_2 \quad \Gamma \vdash t_3}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3} \text{ [IF]}$$
$$\Gamma \vdash \text{zero}$$
$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{iszero } t} \text{ [ISZ]}$$
$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{succ } t} \text{ [SUC]}$$

# Typing rules for booleans and naturals

$$\Gamma \vdash \text{true} : \text{Bool}$$
$$\Gamma \vdash \text{false} : \text{Bool}$$
$$\frac{\Gamma \vdash t_1 \quad \Gamma \vdash t_2 \quad \Gamma \vdash t_3}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3} \text{ [IF]}$$
$$\Gamma \vdash \text{zero}$$
$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{iszero } t} \text{ [ISZ]}$$
$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{succ } t} \text{ [SUC]}$$

# Typing rules for booleans and naturals

$$\Gamma \vdash \text{true} : \text{Bool}$$
$$\Gamma \vdash \text{false} : \text{Bool}$$
$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : \mathbb{T} \quad \Gamma \vdash t_3 : \mathbb{T}}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \mathbb{T}} \text{ [IF]}$$
$$\Gamma \vdash \text{zero}$$
$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{iszero } t} \text{ [ISZ]}$$
$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{succ } t} \text{ [SUC]}$$

# Typing rules for booleans and naturals

$$\Gamma \vdash \text{true} : \text{Bool}$$
$$\Gamma \vdash \text{false} : \text{Bool}$$
$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : \mathbb{T} \quad \Gamma \vdash t_3 : \mathbb{T}}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \mathbb{T}} \text{ [IF]}$$
$$\Gamma \vdash \text{zero} : \text{Nat}$$
$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{iszero } t} \text{ [ISZ]}$$
$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{succ } t} \text{ [SUC]}$$

# Typing rules for booleans and naturals

$$\Gamma \vdash \text{true} : \text{Bool}$$
$$\Gamma \vdash \text{false} : \text{Bool}$$
$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : \text{T} \quad \Gamma \vdash t_3 : \text{T}}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{T}} \text{ [IF]}$$
$$\Gamma \vdash \text{zero} : \text{Nat}$$
$$\frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{iszero } t : \text{Bool}} \text{ [ISZ]}$$
$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{succ } t} \text{ [SUC]}$$

# Typing rules for booleans and naturals

$$\Gamma \vdash \text{true} : \text{Bool}$$
$$\Gamma \vdash \text{false} : \text{Bool}$$
$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} [\text{IF}]$$
$$\Gamma \vdash \text{zero} : \text{Nat}$$
$$\frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{iszero } t : \text{Bool}} [\text{ISZ}]$$
$$\frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{succ } t : \text{Nat}} [\text{SUC}]$$



# Properties of the simply typed $\lambda$ -calculus (1)

## Theorem (Strong normalization) :

In  $\lambda_{\rightarrow}$ , every expression reduces to a value in a finite number of steps.

- It is an example of programming language / model of computation where termination is decidable.
- Hence it is **incomplete**, and cannot express some computable functions. (restricted to the Church naturals, it can only compute extended polynomials)
- **PCF** defined as  $\lambda_{\rightarrow}$  extended with **recursion** and a type for naturals is a Turing-complete language.

# Properties of the simply typed $\lambda$ -calculus (2)

The type system of  $\lambda_{\rightarrow}$  is coherent with regard to  $\beta$ -reduction :

## Theorem (Type preservation) :

If  $t : T$  is typable, and  $t \rightarrow_{\beta} u$ , then  $u : T$  is typable.

## Theorem (Progress) :

If  $t : T$  is typable, then either  $t$  is a value or it can be  $\beta$ -reduced further.

## Definition (Type safety)

A programming language possessing a type system with the preservation and progress properties is said to be **type-safe**.

What is the manifestation of type-safety in classic programming languages?

```
char x    = 12345; // Char
void *px  = &x;    // v
int *py   = px;    // v
int y     = *py;   // Int
```

Non-preservation

```
int a = INT_MIN;
int b = -1;
return a/b;
// → Runtime failure
```

Non-progress

# Types as approximations

Values of a given type are composable and interchangeable.

## Substitution lemma

Given an expression  $t : T$  containing a sub-expression  $x : S$ , then  $x$  can be substituted to any expression  $s$  of type  $S$  without affecting the type of  $t$ .

$$\frac{\Gamma, x : S \vdash t : T \quad \Gamma \vdash s : S}{\Gamma \vdash [x \mapsto s]t : T}$$

# Not every expression is typable

## Limits of type systems : Incompleteness

There exist  $\lambda$ -expressions that are not typable in  $\lambda_{\rightarrow}$ .

### Example

The expression  $nt ::= \lambda x.(x\ x)$  is not typable in  $\lambda_{\rightarrow}$ .

Sketch of proof :

- If  $nt$  were typable,  $x$  would have a type  $T$ .
- Since  $x$  appears on the left of an application,  $T \equiv U \rightarrow V$ .
- But  $x$  also appears on the right of the same application, hence  $T \equiv U$ .
- There is no type in  $\mathbf{Typ}$  such that  $U \equiv U \rightarrow V$ .

# Conservativeness of typing

## Limits of type systems : Conservativeness

A type system is in general **conservative** : there exist expressions in  $\lambda_{\rightarrow}$  that are not typable even though they evaluate safely.

- Simple programs mixing different types of values :

```
let pi = fun b → if b then 3.14 else "Pie";;  
if (pi true > 3.) then print_string (pi false);;
```

- The fixed-point combinator (also called the **Y-combinator**) :

$$Y ::= \lambda f. (\lambda x. f \_ (x \_ x)) \_ (\lambda x. f \_ (x \_ x))$$

... that can be used to encode recursion into the language.

# Partial functions

## Limits of type systems : Liberalness

A type system is in general **liberal** : it cannot discriminate all the stuck expressions of a programming language with simple arithmetic.

Consider the addition of a predecessor function to  $\lambda_{\rightarrow}$  :

$$\frac{\text{pred succ } t \rightarrow_{\beta} t}{\frac{t \rightarrow_{\beta} t'}{\text{pred } t \rightarrow_{\beta} \text{pred } t'}}$$
$$\frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{pred } t : \text{Nat}} \text{ [PRE]}$$

The expression `pred zero` is well-typed and yet stuck. Possible solutions :

- either consider that the evaluation can progress (on floats, return `inf`)
- or add a mechanism that redirects the evaluation (e.g exceptions).

# Designing a language with types

1. Define a programming language as the set of expressions of a grammar.
2. Define an **operational semantics** that performs a computation.
3. Select a set of **values** that are the results of the evaluation.

Usually, the evaluation function cannot be meaningful on the complete language : some expressions remain **stuck**.

4. Set typing rules and restrict the language to **well-typed expressions**.

Type-safety ensures every computation to be either infinite or yield a value.



## Example : handling state

### Syntax and Types

$t ::=$	$\dots$	<i>expressions</i>		
	$()$	<i>unit</i>		
	$\text{ref } t$	<i>reference</i>	$v ::=$	$\dots$ <i>values</i>
	$!t$	<i>dereference</i>		$()$ <i>unit</i>
	$l$	<i>location</i>		$l$ <i>location</i>
	$t := t$	<i>assignment</i>		
	$t; t$	<i>sequence</i>		

- The **locations** are the internal representations of references, i.e the result of the computation of an expression  $\text{ref } t$ .
- The associations between locations and values are saved into a **store**.

## Example : handling state

- A **store**  $\mu$  is a dictionary mapping locations to values :

$$\mu ::= \ell_1 \rightarrow v_1, \dots, \ell_n \rightarrow v_n$$

- The store acts as a **context** and is modified during the evaluation.

### Evaluation rules

$$\frac{\ell \notin \text{dom}(\mu)}{\begin{array}{c} \mu \quad \mu, \ell \rightarrow v \\ \text{ref } v \rightarrow_{\beta} \ell \end{array}}$$

$$\frac{\mu(\ell) = v}{\begin{array}{c} \mu \quad \mu \\ !\ell \rightarrow_{\beta} v \end{array}}$$

$$\begin{array}{c} \mu \quad [\ell \rightarrow v]\mu \\ \ell := v \rightarrow_{\beta} () \end{array}$$

$$\frac{\begin{array}{c} \mu \quad \mu' \\ t_1 \rightarrow_{\beta} v_i \end{array} \quad \begin{array}{c} \mu' \quad \mu'' \\ t_2 \rightarrow_{\beta} v_r \end{array}}{\begin{array}{c} \mu \quad \mu'' \\ t_1; t_2 \rightarrow_{\beta} v_r \end{array}}$$

# Example : handling state

Types

$T ::= \dots$

*unit type*

*reference type*

Typing rules

$$\frac{\Gamma \vdash s \quad \Gamma \vdash t}{\Gamma \vdash s; t}$$

$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{ref } t}$$

$$\frac{\Gamma \vdash r}{\Gamma \vdash !r}$$

$$\frac{\Gamma \vdash r \quad \Gamma \vdash t}{\Gamma \vdash r := t}$$

# Example : handling state

## Types

$T ::=$  ...  
**Unit** *unit type*  
**Ref**[ $T$ ] *reference type*

## Typing rules

$$\frac{\Gamma \vdash s \quad \Gamma \vdash t}{\Gamma \vdash s; t}$$

$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{ref } t}$$

$$\frac{\Gamma \vdash r}{\Gamma \vdash !r}$$

$$\frac{\Gamma \vdash r \quad \Gamma \vdash t}{\Gamma \vdash r := t}$$

# Example : handling state

## Types

$T ::=$  ...  
 $\text{Unit}$  *unit type*  
 $\text{Ref}[T]$  *reference type*

## Typing rules

$$\frac{\Gamma \vdash s : \text{Unit} \quad \Gamma \vdash t : T}{\Gamma \vdash s; t : T}$$

$$\frac{\Gamma \vdash t}{\Gamma \vdash \text{ref } t}$$

$$\frac{\Gamma \vdash r}{\Gamma \vdash !r}$$

$$\frac{\Gamma \vdash r \quad \Gamma \vdash t}{\Gamma \vdash r := t}$$

# Example : handling state

## Types

$T ::=$  ...  
 $\text{Unit}$  *unit type*  
 $\text{Ref}[T]$  *reference type*

## Typing rules

$$\frac{\Gamma \vdash s : \text{Unit} \quad \Gamma \vdash t : T}{\Gamma \vdash s; t : T}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \text{ref } t : \text{Ref}[T]}$$

$$\frac{\Gamma \vdash r}{\Gamma \vdash !r}$$

$$\frac{\Gamma \vdash r \quad \Gamma \vdash t}{\Gamma \vdash r := t}$$

# Example : handling state

## Types

$T ::=$  ...  
 $\text{Unit}$  *unit type*  
 $\text{Ref}[T]$  *reference type*

## Typing rules

$$\frac{\Gamma \vdash s : \text{Unit} \quad \Gamma \vdash t : T}{\Gamma \vdash s; t : T}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \text{ref } t : \text{Ref}[T]}$$

$$\frac{\Gamma \vdash r : \text{Ref}[T]}{\Gamma \vdash !r : T}$$

$$\frac{\Gamma \vdash r \quad \Gamma \vdash t}{\Gamma \vdash r := t}$$

# Example : handling state

## Types

$T ::=$  ...  
 $\text{Unit}$  *unit type*  
 $\text{Ref}[T]$  *reference type*

## Typing rules

$$\frac{\Gamma \vdash s : \text{Unit} \quad \Gamma \vdash t : T}{\Gamma \vdash s; t : T}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \text{ref } t : \text{Ref}[T]}$$

$$\frac{\Gamma \vdash r : \text{Ref}[T]}{\Gamma \vdash !r : T}$$

$$\frac{\Gamma \vdash r : \text{Ref}[T] \quad \Gamma \vdash t : T}{\Gamma \vdash r := t : \text{Unit}}$$



## Example : algebraic datatypes

### Definition (Algebraic Datatype)

An **algebraic datatype** is a type associated to a set of values defined by a regular tree grammar.

### Example : lists containing only integers

$$\text{NatList} \rightarrow \text{Nil} \mid \text{Cons}(\text{Nat}, \text{NatList})$$

**Nil** is a terminal of arity 0, **Cons** is a terminal of arity 2.

In order to introduce such a datatype into the language, it is necessary to :

- add a way to construct the values,
- and another to deconstruct them.

## Example : algebraic datatypes

- Construction : associate to each terminal a keyword acting as a function with the same arity :

```
Nil                                (* Nil is a constant *)  
Cons(2, Nil)                       (* Cons takes 2 arguments *)  
Cons(1, Cons(2, Cons(3, Nil)))    (* their composition yields complex lists *)
```

- Deconstruction / Pattern-matching : associate to each non-terminal a mechanism to select its associated production rules :

```
let length l = case l of           (* selection depending on l being *)  
| Nil          → 0                 (* either Nil *)  
| Cons(x, xs) → 1 + length xs      (* or a Cons with two arguments *)
```

# Example : algebraic datatypes

## Syntax and Types

$t ::= \dots$	<i>expressions</i>
Nil	<i>nil</i>
Cons( $t, t$ )	<i>cons</i>
case $t$ of $\left[ \begin{array}{l} \text{Nil} \rightarrow t \\ \text{Cons}(x, y) \rightarrow t \end{array} \right]$	<i>case</i>

Note that  $x$  and  $y$  in the case-expression are special variable names that cannot be modified in this example.

$v ::= \dots$	<i>values</i>
Nil	<i>nil</i>
Cons( $v, v$ )	<i>cons</i>
$T ::= \dots$	
NatList	<i>list type</i>

# Example : algebraic datatypes

## Evaluation Rules

$$\frac{t_1 \rightarrow_{\beta} t'_1}{\text{Cons}(t_1, t_2) \rightarrow_{\beta} \text{Cons}(t'_1, t_2)}$$

$$\frac{t_1 \rightarrow_{\beta} t'_1}{\text{Cons}(v, t_1) \rightarrow_{\beta} \text{Cons}(v, t'_1)}$$

$$\frac{t_1 \rightarrow_{\beta} t'_1}{\text{case } t_1 \text{ of } \begin{bmatrix} \text{Nil} & \rightarrow t_2 \\ \text{Cons}(x, y) & \rightarrow t_3 \end{bmatrix} \rightarrow_{\beta} \text{case } t'_1 \text{ of } \begin{bmatrix} \text{Nil} & \rightarrow t_2 \\ \text{Cons}(x, y) & \rightarrow t_3 \end{bmatrix}}$$

$$\text{case Nil of } \begin{bmatrix} \text{Nil} & \rightarrow t_1 \\ \text{Cons}(x, y) & \rightarrow t_2 \end{bmatrix} \rightarrow_{\beta} t_1$$

$$\text{case Cons}(v_1, v_2) \text{ of } \begin{bmatrix} \text{Nil} & \rightarrow t_1 \\ \text{Cons}(x, y) & \rightarrow t_2 \end{bmatrix} \rightarrow_{\beta} [x \mapsto v_1, y \mapsto v_2] t_2$$

# Example : algebraic datatypes

## Typing Rules

$$\Gamma \vdash \text{Nil}$$

$$\frac{\Gamma \vdash t_1 \quad \Gamma \vdash t_2}{\Gamma \vdash \text{Cons}(t_1, t_2)}$$

$$\frac{\Gamma \vdash t \quad \Gamma \vdash t_1 \quad \Gamma, x, y \vdash t_2}{\Gamma \vdash \text{case } t \text{ of } \left[ \begin{array}{l} \text{Nil} \rightarrow t_1 \\ \text{Cons}(x, y) \rightarrow t_2 \end{array} \right]}$$

# Example : algebraic datatypes

## Typing Rules

$$\Gamma \vdash \text{Nil} : \text{NatList}$$
$$\frac{\Gamma \vdash t_1 \quad \Gamma \vdash t_2}{\Gamma \vdash \text{Cons}(t_1, t_2)}$$
$$\frac{\Gamma \vdash t \quad \Gamma \vdash t_1 \quad \Gamma, x, y \vdash t_2}{\Gamma \vdash \text{case } t \text{ of } \left[ \begin{array}{l} \text{Nil} \rightarrow t_1 \\ \text{Cons}(x, y) \rightarrow t_2 \end{array} \right]}$$

# Example : algebraic datatypes

## Typing Rules

$$\Gamma \vdash \text{Nil} : \text{NatList}$$
$$\frac{\Gamma \vdash t_1 : \text{Nat} \quad \Gamma \vdash t_2 : \text{NatList}}{\Gamma \vdash \text{Cons}(t_1, t_2) : \text{NatList}}$$
$$\frac{\Gamma \vdash t \quad \Gamma \vdash t_1 \quad \Gamma, x, y \vdash t_2}{\Gamma \vdash \text{case } t \text{ of } \left[ \begin{array}{l} \text{Nil} \rightarrow t_1 \\ \text{Cons}(x, y) \rightarrow t_2 \end{array} \right]}$$

# Example : algebraic datatypes

## Typing Rules

$$\Gamma \vdash \text{Nil} : \text{NatList}$$

$$\frac{\Gamma \vdash t_1 : \text{Nat} \quad \Gamma \vdash t_2 : \text{NatList}}{\Gamma \vdash \text{Cons}(t_1, t_2) : \text{NatList}}$$

$$\frac{\Gamma \vdash t : \text{NatList} \quad \Gamma \vdash t_1 : T \quad \Gamma, x : \text{Nat}, y : \text{NatList} \vdash t_2 : T}{\Gamma \vdash \text{case } t \text{ of } \left[ \begin{array}{l} \text{Nil} \rightarrow t_1 \\ \text{Cons}(x, y) \rightarrow t_2 \end{array} \right] : T}$$



# Summary on the simply-typed lambda-calculus

We showed how to endow a language with a type system and how to perform verifications at a syntactic level.

- Type systems and languages are **modular** and can be extended easily ;
- **Type safety** is a key property for a typed language, ensuring stability properties of programs respecting well-defined bounds ;
- Nevertheless, type systems are by essence both **conservative** and **liberal** in their verifications.

Next, we consider the different decision problems for typed expressions.

# Type checking and inference

Generally the main problems with regard to typing are :

- **Typability** : for an expression  $t$ , is there a type  $T$  and a derivation tree proving that  $t : T$  ?
- **Type checking** : given an expression  $t$ , a type  $T$  and an environment typing the variables of  $t$  (free or bounded), build a derivation tree which proves  $t : T$  or find an inconsistency ;
- **Type inference** : for a typable expression  $t$ , compute a type  $T$  such that there exists a derivation tree which proves  $t : T$ .

In order to solve these problems, we shall :

- derive a system of equations called **constraints** from the expression ;
- compute a solution to this system if any, or prove that there is none.


## Definition (Substitution)

A **substitution**  $\sigma$  is an application from type variables to types. It can be extended as a function from types to types.

### Example

Consider the substitution  $\sigma ::= \{X \mapsto (Y \rightarrow Y), Y \mapsto \text{Nat}\}$ . Then :

- $\sigma(X) = Y \rightarrow Y$ ,  $\sigma(Y) = \text{Nat}$
- $\sigma(Y \rightarrow \text{Bool}) = \text{Nat} \rightarrow \text{Bool}$
- $\sigma \circ \sigma(X) = \text{Nat} \rightarrow \text{Nat}$

- Not very different from the substitutions  defined for expressions.
- Cycles in substitutions should be handled carefully.

### Definition (Type constraints)

A **constraint** is an equation of the form  $S = T$  where  $S, T \in \text{Typ}$ .

A **constraint set**  $\mathcal{C}$  is a finite set of constraints.

### Definition (Unification)

The substitution  $\sigma$  is said to **unify**  $\mathcal{C}$  iff for all equation  $S = T$  in  $\mathcal{C}$ ,  $\sigma S$  and  $\sigma T$  are syntactically equal.

# Example : type checking (1)

```
if b then (f_zero) else (f_true)
```

b : B

f : F

# Example : type checking (1)

```
if b then (f_zero) else (f_true)
```

$b : B$

$f : F$

$B = \text{Bool}$

$F = U \rightarrow V$

# Example : type checking (1)

```
if b then (f_zero) else (f_true)
```

$b : B$

$f : F$

$B = \text{Bool}$

$F = U \rightarrow V$

$U = \text{Bool}$

$U = \text{Nat}$

## Example : type checking (1)

```
if b then (f_zero) else (f_true)
```

$b : B$

$f : F$

$B = \text{Bool}$

$F = U \rightarrow V$

$U = \text{Bool}$

$U = \text{Nat}$

$\Rightarrow$  Type error :  $\text{Bool}$  used where  $\text{Nat}$  expected.



## Example : type checking (2)

```
if b then (f_zero) else (f_succ zero)
```

b : B

f : F

## Example : type checking (2)

if b then (f\_zero) else (f\_succ zero)

b : B

f : F

B = Bool

F =  $U \rightarrow V$

## Example : type checking (2)

if b then (f\_zero) else (f\_succ zero)

b : B

f : F

B = Bool

F =  $U \rightarrow V$

U = Nat

V is unconstrained

## Example : type checking (2)

if b then (f\_zero) else (f\_succ zero)

b : B

f : F

B = Bool

F =  $U \rightarrow V$

U = Nat

V is unconstrained

Type checks : the following substitution unifies the constraints :

$\{B \hookrightarrow \text{Bool}, F \hookrightarrow (U \rightarrow V), U \hookrightarrow \text{Nat}\}$

## Definition (Constrained typing)

To deduce a **constrained typing**  $\Gamma \vdash t : T \mid \mathcal{C}$  means that  $t$  has type  $T$  under the assumptions in  $\Gamma$ , whenever the constraints in  $\mathcal{C}$  are satisfied.

$$\frac{t : T \in \Gamma}{\Gamma \vdash t : T \mid \{\}} \text{ [VAR]}$$

$$\frac{T_1, T_2 \text{ fresh} \quad \Gamma, x : T_1 \vdash u : T_2 \mid \mathcal{C} \quad \mathcal{C}_f ::= \mathcal{C} \cup \{U = T_1 \rightarrow T_2\}}{\Gamma \vdash \lambda x. u : U \mid \mathcal{C}_f} \text{ [ABS]}$$

$$\frac{\Gamma \vdash t : T_1 \mid \mathcal{C}_1 \quad \Gamma \vdash u : T_2 \mid \mathcal{C}_2 \quad \mathcal{C}_f ::= \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_1 = T_2 \rightarrow U\}}{\Gamma \vdash (t\_u) : U \mid \mathcal{C}_f} \text{ [APP]}$$

## Example : deduction of a typing

$$\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_ x)) : \mathbf{T}$$

## Example : deduction of a typing

$$\frac{\{f : T_1\} \vdash \lambda x. (f \_ (f \_ x)) : T_2}{\emptyset \vdash \lambda f. \lambda x. (f \_ (f \_ x)) : T}$$

## Example : deduction of a typing

$$\Gamma ::= \{f : T_1, x : T_3\} \vdash (f\_ (f\_ x)) : T_4$$

$$\{f : T_1\} \vdash \lambda x. (f\_ (f\_ x)) : T_2$$

$$\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_ x)) : T$$



## Example : deduction of a typing

$$\frac{\frac{\Gamma \vdash f : T_5 \quad \Gamma \vdash (f\_x) : T_6}{\Gamma ::= \{f : T_1, x : T_3\} \vdash (f\_ (f\_x)) : T_4}}{\frac{\{f : T_1\} \vdash \lambda x. (f\_ (f\_x)) : T_2}{\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : T}}$$

## Example : deduction of a typing

$$\frac{\frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_5} \quad \frac{}{\Gamma \vdash (f\_x) : T_6}}{\Gamma ::= \{f : T_1, x : T_3\} \vdash (f\_ (f\_x)) : T_4} \quad \frac{}{\frac{\{f : T_1\} \vdash \lambda x. (f\_ (f\_x)) : T_2}{\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : T}}$$

## Example : deduction of a typing

$$\frac{\frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_5} \quad \frac{\Gamma \vdash f : T_7 \quad \Gamma \vdash x : T_8}{\Gamma \vdash (f\_x) : T_6}}{\Gamma ::= \{f : T_1, x : T_3\} \vdash (f\_ (f\_x)) : T_4}$$
$$\frac{\{f : T_1\} \vdash \lambda x. (f\_ (f\_x)) : T_2}{\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : T}$$

# Example : deduction of a typing

$$\begin{array}{c}
 \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_5} \qquad \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_7} \qquad \frac{x : T_3 \in \Gamma}{\Gamma \vdash x : T_8} \\
 \hline
 \Gamma \vdash (f\_x) : T_6 \\
 \hline
 \Gamma ::= \{f : T_1, x : T_3\} \vdash (f\_ (f\_x)) : T_4 \\
 \hline
 \{f : T_1\} \vdash \lambda x. (f\_ (f\_x)) : T_2 \\
 \hline
 \emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : T
 \end{array}$$

# Example : deduction of a typing

$$\begin{array}{c}
 \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_5 \mid \{T_1 = T_5\}} \quad \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_7} \quad \frac{x : T_3 \in \Gamma}{\Gamma \vdash x : T_8} \\
 \hline
 \Gamma \vdash (f \_ x) : T_6 \\
 \hline
 \Gamma ::= \{f : T_1, x : T_3\} \vdash (f \_ (f \_ x)) : T_4 \\
 \hline
 \{f : T_1\} \vdash \lambda x. (f \_ (f \_ x)) : T_2 \\
 \hline
 \emptyset \vdash \lambda f. \lambda x. (f \_ (f \_ x)) : T
 \end{array}$$

# Example : deduction of a typing

$$\begin{array}{c}
 \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_5 \mid \{T_1 = T_5\}} \quad \frac{\frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_7 \mid \{T_1 = T_7\}} \quad \frac{x : T_3 \in \Gamma}{\Gamma \vdash x : T_8}}{\Gamma \vdash (f \_ x) : T_6} \\
 \hline
 \frac{\Gamma ::= \{f : T_1, x : T_3\} \vdash (f \_ (f \_ x)) : T_4}{\frac{\{f : T_1\} \vdash \lambda x. (f \_ (f \_ x)) : T_2}{\emptyset \vdash \lambda f. \lambda x. (f \_ (f \_ x)) : T}}
 \end{array}$$

# Example : deduction of a typing

$$\begin{array}{c}
 \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_5 \mid \{T_1 = T_5\}} \quad \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_7 \mid \{T_1 = T_7\}} \quad \frac{x : T_3 \in \Gamma}{\Gamma \vdash x : T_8 \mid \{T_3 = T_8\}} \\
 \hline
 \Gamma \vdash (f \_ x) : T_6 \\
 \hline
 \Gamma ::= \{f : T_1, x : T_3\} \vdash (f \_ (f \_ x)) : T_4 \\
 \hline
 \{f : T_1\} \vdash \lambda x. (f \_ (f \_ x)) : T_2 \\
 \hline
 \emptyset \vdash \lambda f. \lambda x. (f \_ (f \_ x)) : T
 \end{array}$$

# Example : deduction of a typing

$$\begin{array}{c}
 \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_5 \mid \{T_1 = T_5\}} \quad \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_7 \mid \{T_1 = T_7\}} \quad \frac{x : T_3 \in \Gamma}{\Gamma \vdash x : T_8 \mid \{T_3 = T_8\}} \\
 \hline
 \Gamma \vdash (f\_x) : T_6 \mid \{\dots T_7 = T_8 \rightarrow T_6\} \\
 \hline
 \Gamma ::= \{f : T_1, x : T_3\} \vdash (f\_ (f\_x)) : T_4 \\
 \hline
 \{f : T_1\} \vdash \lambda x. (f\_ (f\_x)) : T_2 \\
 \hline
 \emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : T
 \end{array}$$



# Example : deduction of a typing

$$\begin{array}{c}
 \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_5 \mid \{T_1 = T_5\}} \quad \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_7 \mid \{T_1 = T_7\}} \quad \frac{x : T_3 \in \Gamma}{\Gamma \vdash x : T_8 \mid \{T_3 = T_8\}} \\
 \hline
 \Gamma \vdash (f\_x) : T_6 \mid \{\dots T_7 = T_8 \rightarrow T_6\} \\
 \hline
 \Gamma ::= \{f : T_1, x : T_3\} \vdash (f\_ (f\_x)) : T_4 \mid \{\dots T_5 = T_6 \rightarrow T_4\} \\
 \hline
 \{f : T_1\} \vdash \lambda x. (f\_ (f\_x)) : T_2 \\
 \hline
 \emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : T
 \end{array}$$

# Example : deduction of a typing

$$\begin{array}{c}
 \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_5 \mid \{T_1 = T_5\}} \quad \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_7 \mid \{T_1 = T_7\}} \quad \frac{x : T_3 \in \Gamma}{\Gamma \vdash x : T_8 \mid \{T_3 = T_8\}} \\
 \hline
 \Gamma \vdash (f\_x) : T_6 \mid \{\dots T_7 = T_8 \rightarrow T_6\} \\
 \hline
 \Gamma ::= \{f : T_1, x : T_3\} \vdash (f\_ (f\_x)) : T_4 \mid \{\dots T_5 = T_6 \rightarrow T_4\} \\
 \hline
 \{f : T_1\} \vdash \lambda x. (f\_ (f\_x)) : T_2 \mid \{\dots T_2 = T_3 \rightarrow T_4\} \\
 \hline
 \emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : T
 \end{array}$$

## Example : deduction of a typing

$$\begin{array}{c}
 \frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_5 \mid \{T_1 = T_5\}} \quad \frac{\frac{f : T_1 \in \Gamma}{\Gamma \vdash f : T_7 \mid \{T_1 = T_7\}} \quad \frac{x : T_3 \in \Gamma}{\Gamma \vdash x : T_8 \mid \{T_3 = T_8\}}}{\Gamma \vdash (f\_x) : T_6 \mid \{\dots T_7 = T_8 \rightarrow T_6\}} \\
 \frac{\Gamma ::= \{f : T_1, x : T_3\} \vdash (f\_ (f\_x)) : T_4 \mid \{\dots T_5 = T_6 \rightarrow T_4\}}{\frac{\{f : T_1\} \vdash \lambda x. (f\_ (f\_x)) : T_2 \mid \{\dots T_2 = T_3 \rightarrow T_4\}}{\emptyset \vdash \lambda f. \lambda x. (f\_ (f\_x)) : T \mid \{\dots T = T_1 \rightarrow T_2\}}}
 \end{array}$$

List of constraints :

$$\left\{ \begin{array}{l} T_1 = T_5, T_1 = T_7, T_3 = T_8, \\ T_7 = T_8 \rightarrow T_6, T_5 = T_6 \rightarrow T_4, \\ T_2 = T_3 \rightarrow T_4, T = T_1 \rightarrow T_2 \end{array} \right\}$$

## Example : resolution of a list of constraints

$$\left\{ \begin{array}{l} T_1 = T_5, T_1 = T_7, T_3 = T_8, \\ T_7 = T_8 \rightarrow T_6, T_5 = T_6 \rightarrow T_4, \\ T_2 = T_3 \rightarrow T_4, T = T_1 \rightarrow T_2 \end{array} \right\}$$

Deduction of the constraints :

- $T_1 = T_5 = T_7$

## Example : resolution of a list of constraints

$$\left\{ \begin{array}{l} T_1 = T_5, T_1 = T_7, T_3 = T_8, \\ T_7 = T_8 \rightarrow T_6, T_5 = T_6 \rightarrow T_4, \\ T_2 = T_3 \rightarrow T_4, T = T_1 \rightarrow T_2 \end{array} \right\}$$

Deduction of the constraints :

- $T_1 = T_5 = T_7$
- $T_3 = T_8$

## Example : resolution of a list of constraints

$$\left\{ \begin{array}{l} T_1 = T_5, T_1 = T_7, T_3 = T_8, \\ T_7 = T_8 \rightarrow T_6, T_5 = T_6 \rightarrow T_4, \\ T_2 = T_3 \rightarrow T_4, T = T_1 \rightarrow T_2 \end{array} \right\}$$

Deduction of the constraints :

- $T_1 = T_5 = T_7$
- $T_3 = T_8$
- $T_8 \rightarrow T_6 = T_6 \rightarrow T_4$

## Example : resolution of a list of constraints

$$\left\{ \begin{array}{l} T_1 = T_5, T_1 = T_7, T_3 = T_8, \\ T_7 = T_8 \rightarrow T_6, T_5 = T_6 \rightarrow T_4, \\ T_2 = T_3 \rightarrow T_4, T = T_1 \rightarrow T_2 \end{array} \right\}$$

Deduction of the constraints :

- $T_1 = T_5 = T_7$
- $T_3 = T_8$
- $T_8 \rightarrow T_6 = T_6 \rightarrow T_4$
- $T_4 = T_6 = T_8 = T_3$

## Example : resolution of a list of constraints

$$\left\{ \begin{array}{l} T_1 = T_5, T_1 = T_7, T_3 = T_8, \\ T_7 = T_8 \rightarrow T_6, T_5 = T_6 \rightarrow T_4, \\ T_2 = T_3 \rightarrow T_4, T = T_1 \rightarrow T_2 \end{array} \right\}$$

Deduction of the constraints :

- $T_1 = T_5 = T_7$
- $T_3 = T_8$
- $T_8 \rightarrow T_6 = T_6 \rightarrow T_4$
- $T_4 = T_6 = T_8 = T_3$
- $T = T_1 \rightarrow T_2 = (T_4 \rightarrow T_4) \rightarrow T_4 \rightarrow T_4$



## Example : resolution of a list of constraints

$$\left\{ \begin{array}{l} T_1 = T_5, T_1 = T_7, T_3 = T_8, \\ T_7 = T_8 \rightarrow T_6, T_5 = T_6 \rightarrow T_4, \\ T_2 = T_3 \rightarrow T_4, T = T_1 \rightarrow T_2 \end{array} \right\}$$

Deduction of the constraints :

- $T_1 = T_5 = T_7$
- $T_3 = T_8$
- $T_8 \rightarrow T_6 = T_6 \rightarrow T_4$
- $T_4 = T_6 = T_8 = T_3$
- $T = T_1 \rightarrow T_2 = (T_4 \rightarrow T_4) \rightarrow T_4 \rightarrow T_4$

$$\lambda f. \lambda x. (f \_ (f \_ x)) : (T_4 \rightarrow T_4) \rightarrow T_4 \rightarrow T_4$$

## Definition (Unification algorithm)

$\text{unify}(\mathcal{C})$  takes a list of constraints and returns a substitution :

- $\text{unify}(\{\}) = \text{id}$  the identity on **Typ** ;
- if  $\mathcal{C} ::= \{S = T\} \cup \mathcal{C}'$  then :
  - if  $S = T$  syntactically, return  $\text{unify}(\mathcal{C}')$
  - if  $S$  is a variable  $T$  is a type expression,  
if  $S \in T$ , fail, otherwise return  $\text{unify}([S \mapsto T]\mathcal{C}') \circ [S \rightarrow T]$ ,
  - proceed symetrically if  $S$  is a type expression and  $T$  is a variable
  - if  $S = S_1 \rightarrow S_2$  and  $T = T_1 \rightarrow T_2$ ,  
then return  $\text{unify}(\mathcal{C}' \cup \{S_1 = T_1, S_2 = T_2\})$
  - otherwise fail.

# Principal types

## Theorem (Principal types) :

Given a constraint set  $\mathcal{C}$  for an expression  $e : T$ , the unification algorithm returns a substitution  $\sigma$  that unifies all the constraints.

Moreover,  $\sigma$  is the **most general solution** in the following sense : every unifier  $\tau$  of  $\mathcal{C}$  can be decomposed as  $\tau = \nu \circ \sigma$ .

- $\sigma$  is called the **most general unifier** (or mgu) of the set  $\mathcal{C}$ .
- $\sigma(T)$  yields a type for  $e$  that is called the **principal type** of  $e$ .

# Summary on type checking and inference

In this context, both problems of type checking and type inference are reduced to a single constraint solving problem.

- The description of languages and type systems by sequents is **modular** and extensible ;
- The algorithms for checking and inference are **effective** (quadratic complexity in general) for  $\lambda \rightarrow$  ;
- The sequent description and the algorithms are **tightly linked**, involving the same inductive approach.

Other algorithms may prevail for different type systems, in particular for languages with explicit type annotations.

There is a strong relation between type systems and logics :

### Curry-Howard correspondence

Given a derivation tree proving  $\Gamma \vdash P$  in the propositional calculus, one can construct a well-typed expression  $e$  and a derivation tree  $\Gamma \vdash e : P$  in the simply-typed  $\lambda$ -calculus, and conversely.

types	$\Leftrightarrow$	theorems
expressions	$\Leftrightarrow$	proofs

From this seminal idea stemmed numerous developments in proof theory :

- de Bruijn's **Automath** (1967),
- Martin-Löf's **intuitionistic type theory** (1972),
- Milner's **LCF** (1972)  $\Rightarrow$  **HOL** (1988) and **Isabelle** (1986),
- and Huet and Coquand's **calculus of constructions** (1988)  $\Rightarrow$  Coq

## General tactics

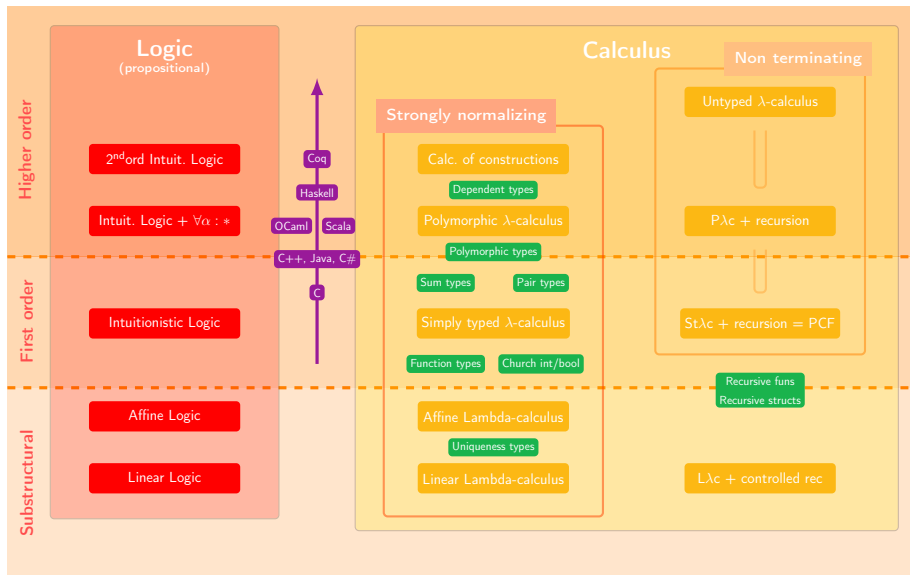
Associate a typed  $\lambda$ -calculus and a logic system.

Constructs in logic are associated to constructs in the calculus :

- The proposition  $A \Rightarrow B$  is associated to the function type  $A \rightarrow B$ .  
“Given an expression/proof of A, I can derive an expression/proof of B”
- The proposition  $A \vee B$  is associated to a sum type  $A \oplus B$ .  
“I contain either an expression/proof of A, or an expression/proof of B”
- The proposition  $A \wedge B$  is associated to a pair type  $A \otimes B$ .  
“I contain both an expression/proof of A, and an expression/proof of B”

And the expressivity of the logic and of the calculus are intertwined.

This is called the **Brouwer-Heyting-Kolmogorov** interpretation for intuitionistic logic (introduced between 1908 and the 1930's).



# Summary on the simply-typed $\lambda$ -calculus

Up until now, the framework we developed around  $\lambda_{\rightarrow}$  contains :

- A language containing functions, integers and booleans, that can be easily extended (cf. references and algebraic data types),
- A family of types **Typ** sufficiently rich to accomodate for all these constructs,
- A framework for type checking and inference within the language.

More importantly, this framework boasts **type-safety** : a type is always an approximation of an expression and remains invariant through evaluation.



# Limits of the simply-typed $\lambda$ -calculus

In some sense, this remains unsatisfactory. Take the following expression :

$$\text{fst} ::= \lambda x. \lambda y. x : X \rightarrow Y \rightarrow X$$

It is sufficiently generic to be reused in different computations :

- `fst_true_zero`
- `fst_(succ zero)_false`

Yet it is **impossible** in  $\lambda_{\rightarrow}$  to **use both** applications in the same program, because it would yield the following contradictory set of constraints :

$$\{X = \text{Bool}, X = \text{Nat}, Y = \text{Bool}, Y = \text{Nat}\}$$

# Let-polymorphism

What we would really like is a **quantification** on the free type variables :

$$\text{fst} ::= \lambda x. \lambda y. x : \forall X, \forall Y, X \rightarrow Y \rightarrow X$$

Intuitively, this just consists in separating the different applications of **fst**, each time replacing the quantified variables by fresh type variables.

$$\text{let fst} = \underbrace{\lambda x. \lambda y. x}_{\forall X, \forall Y, X \rightarrow Y \rightarrow X} \text{ in if b then } \underbrace{(\text{fst\_zero\_true})}_{X_1 = \text{Nat}, Y_1 = \text{Bool}} \text{ else } \underbrace{(\text{fst\_true\_true})}_{X_2 = \text{Bool}, Y_2 = \text{Bool}}$$

This typing process is called the **let-polymorphism**.

# Hindley-Milner type system

1. Extend the type family with type schemes.

**Type scheme** : if  $X_1 \dots X_n$  are type variables and  $T$  in **Typ**,  
then  $\forall X_1 \dots X_n, T$  is a type scheme.

2. Introduce the **let-in** mechanism for the construction of type schemes,

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : \text{generalize}(T_1) \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}$$

3. Allow the application of a “quantified” expression at different places involving potentially different types.

$$\frac{x : \forall X_1 \dots X_n, T \in \Gamma}{\Gamma \vdash x : [X_1 \mapsto Y_1, \dots, X_n \mapsto Y_n]T}$$

# Hindley-Milner type system

The result is the **Hindley-Milner type system**.

- Used in OCaml and the other languages from the ML family.
- It's an example of a language with an **inferred polymorphism**.

```
let fst x y = x;; (* val fst : 'a → 'b → 'a = <fun> *)
```

- A polymorphic function may then be reused in different contexts.

```
let fst x y = x in      (* fst : ∀X, ∀Y, X → Y → X *)  
  fst (1, true) +      (* X = Nat, Y = Bool *)  
  fst (2, "true"));    (* X = Nat, Y = String *)
```

# What can be learned from the H-M type system

- Different type systems entail different families of types, with different sets of properties (ex. : with or without type schemes) ;
- In some systems, the types can be **polymorphic**, and represent sets of concrete types :

$$\forall X, \forall Y, X \rightarrow Y \rightarrow X \quad \equiv \quad \left\{ X \rightarrow Y \rightarrow X, \text{ for } X, Y \in \mathbf{Typ} \right\}$$

- The correspondence between type systems and logics steers the kind of properties we can expect from types (ex : the  $\forall$  quantifiers).

In the following, we explore further the concept of polymorphism.

# Polymorphism

## Polymorphism

An expression in a programming language is said to be **polymorphic** whenever it may be typed with multiply different types.

## Examples

`fst` : `Nat`  $\rightarrow$  `Bool`  $\rightarrow$  `Nat` *or* `Bool`  $\rightarrow$  `Nat`  $\rightarrow$  `Bool` *or* ...

`plus` : `Int`  $\rightarrow$  `Int`  $\rightarrow$  `Int` *or* `Float`  $\rightarrow$  `Float`  $\rightarrow$  `Float` *or* ...

- Applies to functions, but also to non-functional values.
- Polymorphism is a natural property aimed at genericity / code reuse

“Write code once, use it anywhere<sup>1</sup>.”

1. Type conditions may apply.

# Considerations on types

## General idea (Types as sets)

A type represents a set of values.

## Definition (Set of values)

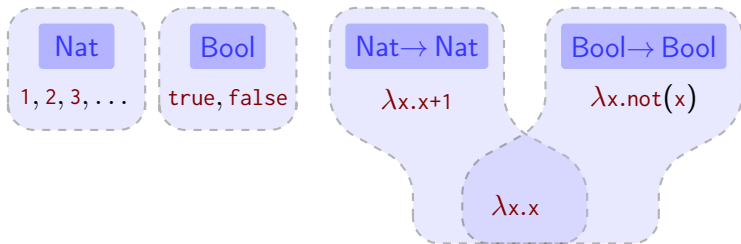
The **set of values** associated to a type  $T$ , noted  $\text{vals}(T)$ , is the set of values of the language that can be typed by  $T$ .

Equivalently,  $e \in \text{vals}(T) \Leftrightarrow e : T$ .

## Definition (Subtype)

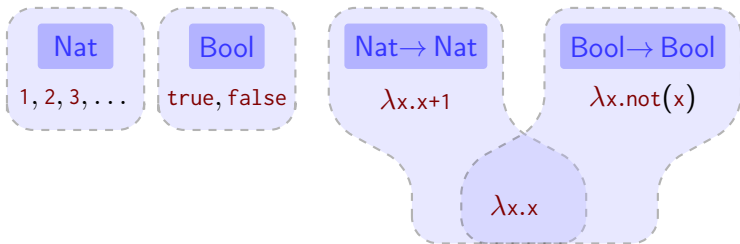
The type  $T$  is said to be a **subtype** of the type  $U$ , noted  $T <: U$ , if and only if  $\text{vals}(T) \subset \text{vals}(U)$ .

- The “types as sets” proposition leads to the following kind of picture :





- The “types as sets” proposition leads to the following kind of picture :



- The identity function  $\text{id} ::= \lambda x.x$  does not have a satisfactory type in  $\lambda_{\rightarrow}$ .
- An extension of  $\lambda_{\rightarrow}$  is the addition of a new type  $T_{\text{id}}$  for  $\text{id}$  such that :

$$\text{vals}(T_{\text{id}}) = \text{vals}(\text{Nat} \rightarrow \text{Nat}) \cap \text{vals}(\text{Bool} \rightarrow \text{Bool})$$

- In this type system,  $\text{id} : T_{\text{id}}$  may be applied either to  $\text{Nat}$  or  $\text{Bool}$  values.

# Properties of subtyping

## Definition (Subsumption rule)

Whenever  $S <: T$ , every expression typable by  $S$  is also typable by  $T$ .

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \text{ [SUB]}$$

## ▶ Extended Substitution Lemma

Consider an expression  $t : T$  containing a free variable  $x : S$ .  
Then  $x$  can be substituted to any expression  $s$  of type  $S' <: S$  without affecting the type of  $t$ .

$$\frac{\Gamma, x : S \vdash t : T \quad \Gamma \vdash s : S' \quad S' <: S}{\Gamma \vdash [x \mapsto s]t : T}$$

## General idea

Create type systems expressing more sophisticated families of types.

Some of these families are well-identified [CW85] :

- Parametric** : define sets of values with the help of universally quantified type parameters (ex :  $\forall X, X \rightarrow X \rightarrow X$ )
- Inclusion** : define sets of values that are related by inclusion or refinement (ex :  $\text{Object} \rightsquigarrow \text{Number} \rightsquigarrow \text{Integer}$ )
- Overloading** : combine sets of values in an adhoc manner, without a particular structure (ex :  $\text{Nat} \oplus \text{Bool}$ )

... while other families do not fit well in that classification  
(cf. for example the Haskell type classes or OCaml open object types)

# Parametric polymorphism (1)

Consider the following extension to our **Typ** family :

## Definition (Parametric types)

Given a type  $T$  containing the variable  $X$ , then  $\forall X, T$  is also a type, called a **parametric** or **universal** type.

The variable  $X$  in  $\forall X, T$  is said to be **bound**. Unbound variables are **free**. A **type scheme** is a parametric type without free variables.

## Example

A type for the first projection  $\text{fst} ::= \lambda x. \lambda y. x$  is  $\forall X, \forall Y, X \rightarrow Y \rightarrow X$

## Parametric polymorphism (2)

Parametric types may be considered as **functions** and be applied :

### Definition (Parametric expression)

Given an expression  $t : T$ , then  $\lambda X.t$  is also an expression, called a **parametric expression** with type  $\forall X, T$ .

A parametric expression  $e : \forall X, T$  can be applied to a type  $U$ , noted  $e[U]$ , which consists in substituting  $X$  by  $U$  inside  $T$ .

This extension introduces a form of computation at the type level :

$$\left( \left[ \lambda X. \lambda Y. (\lambda x : X. (\lambda y : Y. x)) \right] [\text{Int}][\text{Bool}] \right) \_1\_true$$

## Syntax and types

$t ::= \dots$  *expressions*  
 $\lambda X.t$  *type abstraction*  
 $t[T]$  *type application*  
 $v ::= \dots$  *values*  
 $\lambda X.t$  *type abstr. value*  
 $T ::= \dots$  *types*  
 $\forall X, T$  *universal type*


## Typing rules

$$\frac{t \rightarrow_{\beta} t'}{t[T] \rightarrow_{\beta} t'[T]}$$

$$(\lambda X.t)[U] \rightarrow_{\beta} [X \mapsto U]t$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \lambda X.t : \forall X, T}$$

$$\frac{\Gamma \vdash t : \forall X, T}{\Gamma \vdash t[U] : [X \mapsto U]T}$$

Notice the resemblance with the untyped lambda-calculus .

This defines  $\lambda_2$  the **polymorphic** or **2<sup>nd</sup>-order calculus**, or also **System F**.

- It was introduced independently by Girard (1972) and Reynolds (1974).
- It possesses the following properties :

### Theorem (Strong normalization) :

In  $\lambda_2$ , every expression reduces to a value in a finite number of steps.

- It is also **incomplete**, and cannot express all computable functions.  
(restricted to the Church naturals, it can only compute the functions definable in 2<sup>nd</sup>-order Peano arithmetic, among which the Ackermann function)

### Theorem (Impossibility of type inference) :

Type inference in  $\lambda_2$  (without annotations) is undecidable.

# Example : polymorphic lists

Syntax	Evaluation rules	Typing rules
$t ::= \dots$ <i>exprs</i> $\text{nil}$ <i>empty list</i> $\text{cons } t \ t$ <i>cons list</i> $\text{head } t$ <i>list head</i> $\text{tail } t$ <i>list tail</i>	$\frac{t \rightarrow_{\beta} t'}{\text{cons } t \ u \rightarrow_{\beta} \text{cons } t' \ u}$ $\frac{t \rightarrow_{\beta} t'}{\text{cons } v \ t \rightarrow_{\beta} \text{cons } v \ t'}$ $\frac{t \rightarrow_{\beta} t'}{\text{head } t \rightarrow_{\beta} \text{head } t'}$ $\frac{t \rightarrow_{\beta} t'}{\text{tail } t \rightarrow_{\beta} \text{tail } t'}$ $\text{head}(\text{cons } v_1 \ v_2) \rightarrow_{\beta} v_1$ $\text{tail}(\text{cons } v_1 \ v_2) \rightarrow_{\beta} v_2$	$\frac{}{\Gamma \vdash \text{nil} : \text{List}[T]}$ $\frac{\Gamma \vdash x : T \quad \Gamma \vdash l : \text{List}[T]}{\Gamma \vdash \text{cons } x \ l : \text{List}[T]}$ $\frac{\Gamma \vdash l : \text{List}[T]}{\Gamma \vdash \text{head } l : T}$ $\frac{\Gamma \vdash l : \text{List}[T]}{\Gamma \vdash \text{tail } l : \text{List}[T]}$
$v ::= \dots$ <i>values</i> $\text{nil}$ <i>empty list</i> $\text{cons } v \ v$ <i>cons list</i>		
$T ::= \dots$ <i>types</i> $\text{List}[T]$ <i>list type</i>		



## Compare the syntax of polymorphic lists in different languages :

- in Scala :

```
| abstract class List[+A] {  
|   def map[B](f: (A)  $\Rightarrow$  B): List[B] }
```

- in Java :

```
| interface List<E> { E set(int index, E element); }
```

- in C# :

```
| public interface IEnumerable<out T> {  
|   IEnumerable<R> Select<S, R>(this IEnumerable<S> src, Func<S, R> f)
```

- in C++ via iterators :

```
| template<class InputIt, class Function>  
|   Function for_each(InputIt first, InputIt last, Function fn);
```

- in OCaml (`'a list`) and in Haskell (`[a]`)

```
| val map : ('a  $\rightarrow$  'b)  $\rightarrow$  'a list  $\rightarrow$  'b list
```

# Boehm-Berarducci encoding of lists

As a matter of fact, there exists a general technique to encode algebraic types such as the lists in  $\lambda_2$ , called the **Boehm-Berarducci encoding** :

$$\text{List}[T] ::= \forall X, \underbrace{(T \rightarrow X \rightarrow X)}_{\text{cons}} \rightarrow \underbrace{X}_{\text{nil}} \rightarrow X$$

- The empty list is the following value :

$$\text{nil} ::= \left| \lambda X. \lambda c : (T \rightarrow X \rightarrow X). \lambda n : X \right|. n$$

- Consider  $x : T$  and  $xs : \text{List}[T]$ . The list beginning with  $x$  and ending with  $xs$  is the following value :

$$\text{cons } x \text{ } xs ::= \left| \lambda X. \lambda c : (T \rightarrow X \rightarrow X). \lambda n : X \right|. c \ x \ (xs[X] \ c \ n)$$

# Type substitution

## Definition (Type substitution)

A **type substitution**  $\sigma$  is a finite mapping from type variables to types. We write  $[X_i \mapsto T_i]$  for the substitution mapping  $X_i$  to  $T_i$ .

The **application** of  $\sigma$  to a type  $U \equiv \forall X_1, \dots, \forall X_n, T$ , noted  $\sigma U$  consists in substituting the free occurrences of  $X_i$  inside  $T$  by  $T_i$  simultaneously, and then generalizing the type. Variables bound inside  $T$  are left invariant.

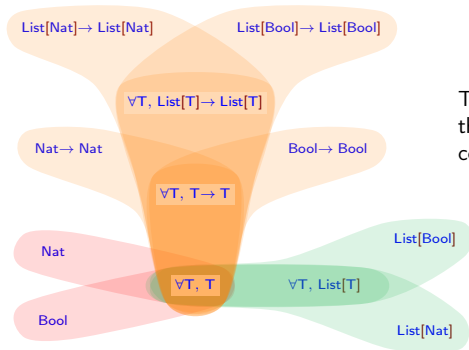
## Examples

- $[X \mapsto \text{Nat}, Y \mapsto \text{Bool}](\forall X Y, X \rightarrow Y \rightarrow X) = \text{Nat} \rightarrow \text{Bool} \rightarrow \text{Nat}$
- $[X \mapsto Z \rightarrow Z](\forall X, X \rightarrow X) = (Z \rightarrow Z) \rightarrow Z \rightarrow Z$

# Parametric subtyping

Lemma (Parametric subtyping) :

For all substitutions  $\sigma$  and all types  $T$ ,  $T <: \sigma T$ .



The set of values typed by  $\text{Bool} \rightarrow \text{Bool}$  contains the set of values typed by  $\forall T, T \rightarrow T$ , which also contains those typed by  $\forall T, \text{List}[T]$ .

# Principal types

What is the “best type” for a given expression ?

## Definition (Principal type)

Given a typable expression  $e$ , the **principal type** of  $e$  is the maximum type  $T$  for the subtype relation (when it exists) such that  $e : T$ .

- Our inference algorithm infers types that are principal for the Hindley-Milner type system.
- The System F type system does **not** have principal types.

$$t ::= \lambda f. \text{if } f(\text{true}) \text{ then } f(1) \text{ else } f(0) \quad \begin{cases} t_1 : (\forall X, X \rightarrow X) \rightarrow \text{Nat} \\ t_2 : (\forall X, X \rightarrow \text{Bool}) \rightarrow \text{Bool} \end{cases}$$

# Implementations of the parametric polymorphism

To compile (parametric) polymorphic code ...

```
class HashTbl<Key, Val extends Object> {  
  HashTbl() { .. }  
  Val get(Key k) { .. }  
  Val put(Key k, Val v) { .. }  
}
```

# Implementations of the parametric polymorphism

To compile (parametric) polymorphic code, two main strategies coexist :

- **Homogeneous translation** : generate code where the generic data has a uniform representation, independent of its real type.

```
class HashTbl<Key, Val extends Object> {  
    HashTbl() { .. }  
    Val get(Key k) { .. }  
    Val put(Key k, Val v) { .. }  
}
```



```
class HashTbl {  
    HashTbl() { .. }  
    Object get(Object k) { .. }  
    Object put(Object k, Object v) { }  
}
```

Example : Java with **type erasure**.

# Implementations of the parametric polymorphism

To compile (parametric) polymorphic code, two main strategies coexist :

- **Heterogeneous translation** : duplicate and tailor the generated code for each possible concrete type effectively used.

```
class HashTbl<Key, Val extends Object> {  
  HashTbl() { .. }  
  Val get(Key k) { .. }  
  Val put(Key k, Val v) { .. }  
}
```

```
class HashTbl_Int_String {  
  HashTbl_Int_String() { .. }  
  string get(int k) { .. }  
  string put(int k, string v) { .. }
```

```
class HashTbl_Char_File {  
  HashTbl_Char_File() { .. }  
  File get(char k) { .. }  
  File put(char k, File v) { .. }
```

Example : C++ via the preprocessor, Rust monomorphism.

The most general approach is a combination of both styles.



# Summary on the parametric polymorphism

- The parametric polymorphism allows the definition of types as logic formulas containing **universally** quantified variables.

$$\forall T, F[T] \equiv \bigcap_{U \in \mathbf{Typ}} F[U]$$

- Type substitution on these variables induces subtyping relations.

$$\forall T, F[T] <: F[U] \text{ where } U \text{ concrete}$$

- For a polymorphic expression, the maximal type wrt subtyping is called the **principal** type. Not all type systems possess maximal types.
- The **inference** of parametric types is possible in the Hindley-Milner type system, but undecidable in System F.

Sometimes, a polymorphic type may be **too generic**.

Take the example of an equality function, with the following type :

$$\forall T, T \rightarrow T \rightarrow \text{Bool}$$

Yet **not all** values are comparable, for instance functional values.

It is natural to restrict the possible **T**s to a family of types.

$$\forall T \in \text{Comparable}, T \rightarrow T \rightarrow \text{Bool}$$

This is a form of **constrained polymorphism**, appearing as :

- the ''a **equality types** in SML, a subset of the types of the language,
- the **Eq** a **type class** in Haskell, defined by a form of overloading,
- the **Comparable<T>** **interface** in Java, with inclusion polymorphism.

# Constrained polymorphism example : equality

In OCaml :

```
let rec belongs x l = match l with  
| []      → false  
| y::ys   → (x=y) || (belongs x ys)
```

```
val belongs : 'a→'a list→bool
```

`belongs sin [cos]` compiles, but  
yields an **exception at runtime**.

```
(* Exception: Invalid_argument  
   "equal: functional value". *)
```

In Haskell :

```
belongs x []      = False  
belongs x (y:ys) = (x==y) ||  
                    (belongs x ys)
```

```
belongs :: Eq t => t→[t]→Bool
```

The expression `belongs sin [cos]`  
simply **does not compile**.

```
No instance for (Eq (a0 → a0))  
arising from a use of "=="
```

# Constrained polymorphism : numeric classes

## Definition (Type class)

In Haskell, a **type class** is a set of concrete types sharing a common generic interface. These concrete types are then **instances** of the type class.

## Example :

```
class Eq a where
    (==) :: a → a → Bool
    (/=) :: a → a → Bool

instance Eq Int where
    (==) i j = -- specific code
    (/=) i j = not (i == j)
```

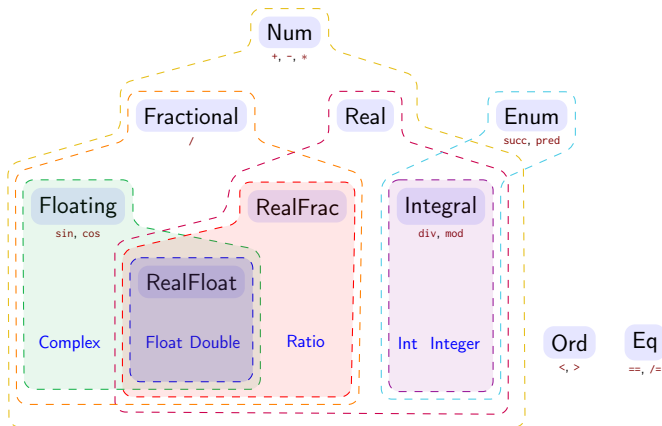
- A type class such as `Eq a` represents the following set of types :

$$\text{Eq}[T] ::= \{T \in \text{Typ}, T \text{ "can be used with" } ==\}$$

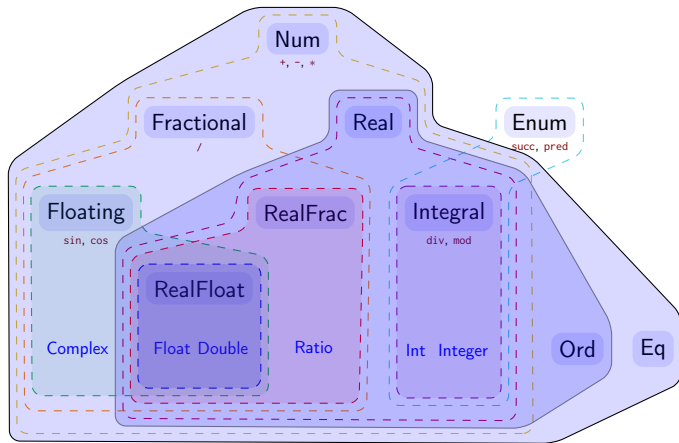
- It is used as a **universal type variable** in the type of `belongs` :

$$\text{belongs} : \forall T \in \text{Eq}[T], T \rightarrow \text{List}[T] \rightarrow \text{Bool}$$

The numeric types in Haskell inherit a structure from the **type classes**.



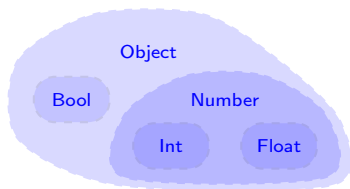
The numeric types in Haskell inherit a structure from the **type classes**.




In the following, we investigate the inclusion relations of sets of values.

# Inclusion polymorphism

Inclusion polymorphism is based on the construction of sets of values sharing relations of inclusion.



Whereas parametric polymorphism defines inclusions **bottom-up** , inclusions in this polymorphism are defined **top-down**.

For a better understanding of these relations, we introduce a new type.

# Record types

## Definition (Record types)

Given a set  $\{l_i\}$  of labels and a set  $\{t_i\}$  of expressions of the same size  $n$ , a **record value** is defined as the expression  $\{l_1 = t_1, \dots, l_n = t_n\}$ .

The pairs  $(l_i, t_i)$  are called the **fields** of the record.

The **projection** of  $r$  onto one of its fields  $(l_i, t_i)$ , noted  $r \triangleright l_i$ , evaluates to  $t_i$ .

The **type** of  $r$  is the set of the types of its fields, noted  $\{l_1 : T_1, \dots, l_n : T_n\}$ .

## Examples

- $\{first = \text{"Haskell"}, last = \text{"Curry"}\} : \{first : \text{String}, last : \text{String}\}$
- $\{hd = 1, tl = \{hd = 2, tl = \{\}\}\} : \{hd : \text{Nat}, tl : \{hd : \text{Nat}, tl : \{\}\}\}$



## Syntax

$t ::= \dots$  *expressions*  
 $\{l_i = t_i\}_{i \in [1; n]}$  *record*  
 $t \triangleright l$  *projection*

$v ::= \dots$  *values*  
 $\{l_i = v_i\}_{i \in [1; n]}$  *record value*

$T ::= \dots$  *types*  
 $\{l_i : T_i\}_{i \in [1; n]}$  *record type*

## Evaluation rules

$$\{l_i = v_i\}_{i \in [1; n]} \triangleright l_j \rightarrow_{\beta} v_j$$

$$\frac{t \rightarrow_{\beta} t'}{t \triangleright l \rightarrow_{\beta} t' \triangleright l}$$

$$\frac{t_j \rightarrow_{\beta} t'_j}{\{\dots l_j = t_j \dots\} \rightarrow_{\beta} \{\dots l_j = t'_j \dots\}}$$

## Typing Rules

$$\frac{\text{for each } i, \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i\} : \{l_i : T_i\}}$$

$$\frac{\Gamma \vdash \{l_i = t_i\} : \{l_i : T_i\}}{\Gamma \vdash t \triangleright l_i : T_i}$$

# All-purpose records

Records are good examples of the saying “he who can do more, can do less”. Consider the following function :

$$\text{half\_size} ::= \lambda r. (r \mapsto \text{size} / 2)$$

It can be happily applied to every record possessing a field *size*.

$r_1 ::= \{\text{size} = 2\}$	$\text{half\_size}(r_1) \rightarrow_{\beta} 1$
$r_2 ::= \{\text{size} = 6, \text{name} = \text{"Alonzo"}\}$	$\text{half\_size}(r_2) \rightarrow_{\beta} 3$
$r_3 ::= \{\text{size} = 2, \text{contents} = \text{Cons}(1, \text{Cons}(2, \text{Nil}))\}$	$\text{half\_size}(r_3) \rightarrow_{\beta} 1$

A natural typing for this function is  $\text{half\_size} : \{\text{size} : \text{Nat}\} \rightarrow \text{Nat}$ . Yet it is too restrictive : it only authorizes the first application. So what ?

# Inclusion subtyping

## Lemma (Record subtyping) :

Let  $T = \{l_i : T_i\}$  and  $T' = \{m_i : T'_i\}$  be two record types.

- **Width subtyping** : if  $T \supset T'$  as sets of pairs labels/types, then  $T <: T'$ .
- **Depth subtyping** : if  $T$  and  $T'$  share exactly the same labels and  $\forall i, T_i <: T'_i$ , then  $T <: T'$ .

## Examples

- **Width** :  $\{size : \text{Nat}, name : \text{String}\} <: \{size : \text{Nat}\}$
- **Depth** : if  $\text{Int} <: \text{Number}$ , then  $\{size : \text{Int}\} <: \{size : \text{Number}\}$

With the following definitions :

- $\text{half\_size} ::= \lambda r. (r \rightarrow \text{size} / 2)$
- $\text{person} ::= \{ \text{size} = 7, \text{name} = \text{"Haskell"} \}$

Let  $\Gamma ::= \{ \text{half\_size} : \{ \text{size} : \text{Nat} \} \rightarrow \text{Nat}, \text{person} : \{ \text{size} : \text{Nat}, \text{name} : \text{String} \} \}$

Then the expression  $(\text{half\_size\_person})$  is well-typed :

$$\frac{\frac{\text{half\_size} \in \Gamma}{\Gamma \vdash \text{half\_size} : \{ \text{size} : \text{Nat} \} \rightarrow \text{Nat}} \quad \frac{\frac{\text{person} \in \Gamma}{\Gamma \vdash \text{person} : \{ \text{size} : \text{Nat}, \text{name} : \text{String} \}}}{\Gamma \vdash \text{person} : \{ \text{size} : \text{Nat} \}}}{\Gamma \vdash (\text{half\_size\_person}) : \text{Nat}}$$

In this case, the subtyping rules solve the “do-more, do-less” problem.  
But what implications does this have on our sets of values?

# Existential types

## Proposition

A record type is by nature an existential type.

All records having a field *size* of type `Nat` can be typed with  $\{size : Nat\}$ .

$$\begin{aligned}\{size : Nat\} &\equiv \bigcup_T \{size : Nat\} \cup T \\ &\equiv \exists T. \{size : Nat\} \cup T\end{aligned}$$

## Example

`half_size` :  $(\exists T. \{size : Nat\} \cup T) \rightarrow Nat$


## Aside : upcast, downcast

### Definition (Casting)

**Casting** (or ascription) consists in ascribing a particular type to an expression in an explicit manner. It has no effect on the value.

The expression  $v$  as  $T$  is called a cast from  $v$  into the type  $T$ .

$$v \text{ as } T \rightarrow_{\beta} v \qquad \frac{\Gamma \vdash t : ?}{\Gamma \vdash t \text{ as } T : T}$$

- A cast can be seen as an operation redefining the type of an expression.
- Casting into a supertype is also called an **upcast**.  
Upcasts are implicit with the  subsumption rule.
- Casting into a subtype is also called an **downcast**.  
Downcasts are usually checked dynamically.

$$\frac{\Gamma \vdash t : S \quad S \leq T}{\Gamma \vdash t \text{ as } T : T}$$

$$\frac{\mathcal{E} \vdash v : T}{v \text{ as } T \rightarrow_{\beta} v}$$

Consider the following function :

$$\text{cut\_in\_half} ::= \lambda r. \left\{ r \rightarrow \text{size} := (\text{half\_size} \_ (r \rightarrow \text{size})); r \right\}$$

It basically takes a record with a *size* field and returns a record where this field has been modified and the others left untouched.

What is a good type for *cut\_in\_half* ?

Consider the following function :

$$\text{cut\_in\_half} ::= \lambda r. \left\{ r \triangleright \text{size} := (\text{half\_size} \_ (r \triangleright \text{size})); r \right\}$$

It basically takes a record with a *size* field and returns a record where this field has been modified and the others left untouched.

What is a good type for *cut\_in\_half* ?

- $\text{cut\_in\_half} : \forall T, T \rightarrow T$  ?

Too generic, no way of ensuring the existence of the *size* field.

- $\text{cut\_in\_half} : \{ \text{size} : \text{Ref}[\text{Nat}] \} \rightarrow \{ \text{size} : \text{Ref}[\text{Nat}] \}$  ?

Too restrictive, the return type constrains the result.

Same behavior as the *clone* method in Java, requiring downcasts.



Consider the following function :

$$\text{cut\_in\_half} ::= \lambda r. \left\{ r \triangleright \text{size} := (\text{half\_size\_}(r \triangleright \text{size})); r \right\}$$

It basically takes a record with a *size* field and returns a record where this field has been modified and the others left untouched.

What is a good type for *cut\_in\_half* ?

What about :  $\forall T, \left( \{ \text{size} : \text{Nat} \} \cup T \right) \rightarrow \left( \{ \text{size} : \text{Nat} \} \cup T \right)$  ✓

Or is it an existential type ?  $\left( \exists T. \{ \text{size} : \text{Nat} \} \cup T \right) \rightarrow \left( \exists U. \{ \text{size} : \text{Nat} \} \cup U \right)$  ✗  
 $\exists T. \left( \{ \text{size} : \text{Nat} \} \cup T \right) \rightarrow \left( \{ \text{size} : \text{Nat} \} \cup T \right)$  ✗

# Aside on existentials and universals

**Beware :**  $\forall T, \{size : Nat\} \cup T \neq \exists T. \{size : Nat\} \cup T$

A universal type  $T$  can substitute for **all** possible types.

An existential type  $T$  can substitute for only **one**.

Nevertheless, there are equivalences :

## Equivalence theorem

$$\left( \exists x. P(x) \right) \Rightarrow Q \quad \Leftrightarrow \quad \forall x. \left( P(x) \Rightarrow Q \right)$$

## Example

$$\left( \exists T. \{size : Nat\} \cup T \right) \rightarrow Nat \equiv \forall T, \left( \{size : Nat\} \cup T \rightarrow Nat \right)$$

Naming the existential variables prompts for **more precise types** :

- Existential types in PureScript :

```
cut_in_half :: forall b. { size :: Int | b } → { size :: Int | b }
```

- Open types in OCaml :

```
cut_in_half : (<get_size : int; set_size : int → unit; ..> as 'a) → 'a
```

But in general, libraries contain few functions requiring such types.

# Existentials as an abstraction means

## Definition (Abstract data type)

An **abstract data type** or ADT consists of :

- a type variable  $T$  and a set of operation types acting on values of type  $T$ .
- a concrete type  $S$  and an implementation of these operation types where the variable  $T$  is substituted by  $S$ .

Akin to interfaces in Java or module signatures in ML.

```
interface Counter {  
  
    int get();  
    Counter incr();  
}
```

```
class CImpl implements Counter {  
    private int val = 0;  
    CImpl(int in) { val = in; }  
    int get()      { return val; }  
    Counter incr() { return new CImpl(val+1); }  
}
```

The existential type corresponds here to the “abstract” part of the ADT.

# Existentials as an abstraction means

## Definition (Abstract data type)

An **abstract data type** or ADT consists of :

- a type variable  $T$  and a set of operation types acting on values of type  $T$ .
- a concrete type  $S$  and an implementation of these operation types where the variable  $T$  is substituted by  $S$ .

Akin to interfaces in Java or module signatures in ML.

```
module type COUNTER = sig
  type counter
  val new_c : unit → counter
  val get   : counter → int
  val inc   : counter → counter
end
```

```
module C : COUNTER = struct
  type counter = int
  let new_c () = 0
  let get c = c
  let inc c = c + 1
end
```

The existential type corresponds here to the “abstract” part of the ADT.

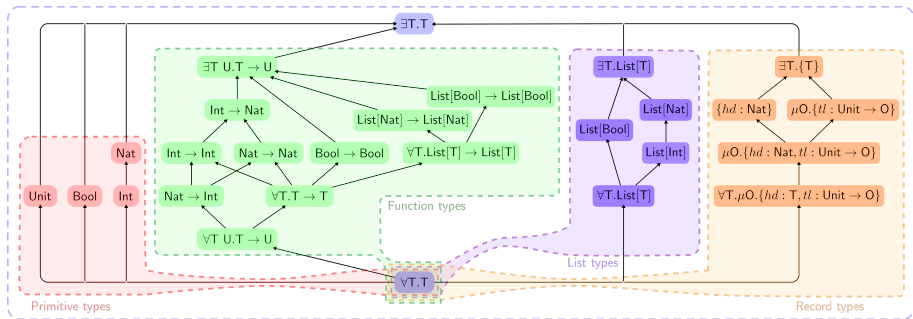
# Existentials vs Universals

- Existentials are items of **abstraction**.  
Allowing multiple implementations,  
Offering a precise protocol of exchange.
- Universals are items of **genericity**.  
Maximising code reuse,  
With few knowledge on the values they manipulate.

In that, the combination of both polymorphisms is natural.

```
<T extends Comparable<T>> void sort(List<T> list)
```

$$\forall T, \exists U. \text{List}[\text{Comparable}[T] \cup U] \rightarrow \text{Unit}$$

# Object types (1)

Objects can be modelled as records with access to `self` : they are **recursive**.

```
let (c:cpt) = let rec self = {   (* Recursive definition *)
    v    = 0;
    get  = (fun () → self.v);
    set  = (fun y → self.v ← y);
    inc  = (fun () → self.set (self.get() + 1)); }
  in self;;
```

In OCaml, classes are indeed identified to their **recursive** constructors :

```
class cpt = fun init → object (self)
    val mutable v:int = init
    method get      = v
    method set d    = v ← d
    method inc ()   = self#set (self#get + 1)
end;;
```



## Object types (2)

- A class, representing a set of values, can be identified to a type.
- Accordingly, these types are also recursive :

$$\text{Object} ::= \text{fix}(\lambda T. \{ \begin{array}{l} \text{equals} : T \rightarrow \text{Bool}, \\ \text{clone} : \text{Unit} \rightarrow T \end{array} \})$$

... where **fix** is a fixed-point operator.

- This definition yields an **infinite** type represented by a rewriting rule.
- Usually, this fixed-point is made invisible by nominal types :

$$\text{Object} ::= \{ \begin{array}{l} \text{equals} : \text{Object} \rightarrow \text{Bool}, \\ \text{clone} : \text{Unit} \rightarrow \text{Object} \end{array} \}$$

# Summary on the inclusion polymorphism

- The inclusion polymorphism allows the definition of types by refining sets of values into more specific subsets.
- These sets of values can be identified as logic formulas containing **existentially** quantified variables.

$$\exists T. F[T] \equiv \bigcup_{U \in \text{Typ}} F[U]$$

- Subtyping relations with existential types promote **abstraction** by masking concrete types :

$$F[U] <: \exists T. F[T] \text{ where } U \text{ concrete}$$

- Object types are at the same time existential and recursive types.

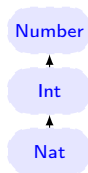
# Aside on Inheritance

## Inheritance

**Inheritance** is a mechanism to derive new classes from old ones by :

- (i) adding implementation for new methods
- (ii) and / or **overriding** implementations of old methods.

Consider a method `m` defined in `Number` and overridden in its subclasses.



```
class Number { m(..); }
```

`m(x : Number, ...)`

```
class Int extends Number { m(..); }
```

`m(x : Int, ...)`

```
class Nat extends Int { m(..); }
```

`m(x : Nat, ...)`

The method `m` can be considered as a function whose definition is **selected** depending on the value of its first parameter.

# Overloading

## Overloading

**Overloading** is a mechanism allowing the use of a single identifier for the representation of multiple values, distinguished according to their type.

### Example : string concatenation in C++

```
string operator+ (const string& lhs, const string& rhs);  
string operator+ (const string& lhs, const char*   rhs);  
string operator+ (const char*   lhs, const string& rhs);  
string operator+ (const string& lhs, char        rhs);  
string operator+ (char          lhs, const string& rhs);
```

### Example : default values in Haskell

```
class Default a where def :: a    -- | The default value for this type  
instance Default Int where def = 0  
instance Default [a] where def = []  
instance (Default a, Default b) => Default (a, b) where def = (def, def)
```

- A manner to represent overloaded values consists in **packing** all the implementations together in a single object.

<code>def ::= 0 ⊕ []</code>	packing 0 and [] together
<code>plus ::= plus<sub>Int</sub> ⊕ plus<sub>List</sub></code>	packing addition and concatenation

- For typing purposes, an overloaded value possesses the types of **all** the values it merges :

- A manner to represent overloaded values consists in **packing** all the implementations together in a single object.

$\text{def} ::= 0 \oplus []$                       packing 0 and [] together  
 $\text{plus} ::= \text{plus}_{\text{Int}} \oplus \text{plus}_{\text{List}}$       packing addition and concatenation

- For typing purposes, an overloaded value possesses the types of **all** the values it merges : an **intersection type**.

$\text{def} : \text{Nat} \ \& \ \forall A, \text{List}[A]$

- It is **not** a union type : as a value,  $\text{def}$  has the possibility to be used indifferently as a number **and** as a list (but only one at a time).

## Consequence

If an identifier is overloaded, a correct implementation must be **selected** every time the identifier is used (at compile-time or at runtime)

Syntax	Evaluation rules	Typing rules
$t ::= \dots$ <i>expressions</i> $t \oplus t$ <i>merge</i>	$\frac{t_1 \rightarrow_{\beta} t'_1}{t_1 \oplus t_2 \rightarrow_{\beta} t'_1 \oplus t_2}$	$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 \oplus t_2 : T_1 \& T_2}$
$T ::= \dots$ <i>types</i> $T \& T$ <i>intersec. type</i>	$\frac{t_2 \rightarrow_{\beta} t'_2}{t_1 \oplus t_2 \rightarrow_{\beta} t_1 \oplus t'_2}$	$\frac{\Gamma \vdash t : T_1 \& T_2}{\Gamma \vdash t : T_1}$
	$v_1 \oplus v_2 \rightarrow_{\beta} v_1$	$\frac{\Gamma \vdash t : T_1 \& T_2}{\Gamma \vdash t : T_2}$
	$v_1 \oplus v_2 \rightarrow_{\beta} v_2$	

- Caution : these rules **break** the type safety of the system.
- At least, a mechanism must be introduced to ensure that the value used at runtime is compatible with the type checked at compile-time.

### Lemma (Intersection subtyping) :

If  $T_1$  and  $T_2$  are two different types,  $T_1 \& T_2 <: T_1$  and  $T_1 \& T_2 <: T_2$

# Some tactics to select an overloaded method (1)

Let  $\text{def} ::= 1 \oplus \text{true} : \text{Bool} \& \text{Nat}$  be an overloaded value.

How can we evaluate the expression :  $\text{if def then } 0 \text{ else def} ?$



# Some tactics to select an overloaded method (1)

Let  $\text{def} ::= 1 \oplus \text{true} : \text{Bool} \& \text{Nat}$  be an overloaded value.

How can we evaluate the expression :  $\text{if def then } 0 \text{ else def} ?$

- Decide the implementation used at **compile-time** (cf. Haskell),

$$\frac{\frac{\frac{\Gamma \vdash \text{def} : \text{Bool} \& \text{Nat}}{\Gamma \vdash \text{def} : \text{Bool}}}{\Gamma \vdash \text{def as Bool} : \text{Bool}} \quad \Gamma \vdash 0 : \text{Nat} \quad \frac{\frac{\frac{\Gamma \vdash \text{def} : \text{Bool} \& \text{Nat}}{\Gamma \vdash \text{def} : \text{Nat}}}{\Gamma \vdash \text{def as Nat} : \text{Nat}}}{\Gamma \vdash \text{if def as Bool then } 0 \text{ else def as Nat} : \text{Nat}}$$

# Some tactics to select an overloaded method (1)

Let  $\text{def} ::= 1 \oplus \text{true} : \text{Bool} \& \text{Nat}$  be an overloaded value.

How can we evaluate the expression :  $\text{if def then } 0 \text{ else def} ?$

- Decide the implementation used at **runtime** (cf. Python),

$$\frac{\mathcal{E} \vdash \text{def} : \text{Bool} \& \text{Nat}}{\text{def} \rightarrow_{\beta} \text{def as Bool}} \quad \text{if def then } 0 \text{ else def} \rightarrow_{\beta} \text{if def as Bool then } 0 \text{ else def} \rightarrow_{\beta} \dots$$

# Some tactics to select an overloaded method (1)

Let  $\text{def} ::= 1 \oplus \text{true} : \text{Bool} \& \text{Nat}$  be an overloaded value.

How can we evaluate the expression :  $\text{if def then } 0 \text{ else def} ?$

- Decide the implementation used at **compile-time** (cf. Haskell),

$$\frac{\frac{\Gamma \vdash \text{def} : \text{Bool} \& \text{Nat}}{\Gamma \vdash \text{def} : \text{Bool}} \quad \Gamma \vdash 0 : \text{Nat} \quad \frac{\Gamma \vdash \text{def} : \text{Bool} \& \text{Nat}}{\Gamma \vdash \text{def} : \text{Nat}}}{\Gamma \vdash \text{if def as Bool then } 0 \text{ else def as Nat} : \text{Nat}}$$

- Decide the implementation used at **runtime** (cf. Python),

$$\frac{\mathcal{E} \vdash \text{def} : \text{Bool} \& \text{Nat}}{\text{def} \rightarrow_{\beta} \text{def as Bool}}{\text{if def then } 0 \text{ else def} \rightarrow_{\beta} \text{if def as Bool then } 0 \text{ else def} \rightarrow_{\beta} \dots}$$

- ... or use a combination of both (cf. Java, C++).

# Static / Late binding

The presence of subtyping allows the following technique :

## Static / Late binding

To every object value is attached a type called its **concrete type**. It may differ from the **apparent type** of this same value in an expression.

At the callpoint of an overloaded method, the appropriate code is selected.

In **static binding**, the selection depends on the **apparent type**.

In **late binding**, the selection depends on the **concrete type**.

## Example

```
String s = new String("Concrete") // Apparent : String / Concrete : String
Object o = (Object) s;           // Apparent : Object / Concrete : String
o.equals(s);                     // Which equals method is called ?
```

Consider the following examples based on a **Counter** class :

```
class Counter {  
    int v; // count calls to inc  
  
    public Counter(int v) { this.v = v; };  
    int get() { return v; }  
    void set(int v) { this.v = v; }  
    void inc() { this.set(this.get() + 1); }  
}
```

```
1 class CounterExt extends Counter {  
2  
3     int a; // count calls to set  
4  
5     public CounterExt(int v) { super(v); a = 0;};  
6     // inherit get  
7     void set(int v) { this.a++; super.set(v); }  
8     // inherit inc  
9     int get_a() { return a; }  
10 }
```

Consider the following examples based on a **Counter** class :

```
class Counter {  
    int v; // count calls to inc  
  
    public Counter(int v) { this.v = v; };  
    int get() { return v; }  
    void set(int v) { this.v = v; }  
    void inc() { this.set(this.get() + 1); }  
}
```

```
1 class CounterExt extends Counter {  
2  
3     int a; // count calls to set  
4  
5     public CounterExt(int v) { super(v); a = 0;};  
6     // inherit get  
7     void set(int v) { this.a++; super.set(v); }  
8     // inherit inc  
9     int get_a() { return a; }  
10 }
```

Consider the call to **set** inside the **inc** method in **Counter** :

- In **early binding**, this call is attached to the apparent type.  
For an object of type **CounterExt**, the **set** method of **Counter** is used.

Consider the following examples based on a **Counter** class :

```
class Counter {  
    int v; // count calls to inc  
  
    public Counter(int v) { this.v = v; };  
    int get() { return v; }  
    void set(int v) { this.v = v; }  
    void inc() { this.set(this.get() + 1); }  
}
```

```
1 class CounterExt extends Counter {  
2  
3     int a; // count calls to set  
4  
5     public CounterExt(int v) { super(v); a = 0;};  
6     // inherit get  
7     void set(int v) { this.a++; super.set(v); }  
8     // inherit inc  
9     int get_a() { return a; }  
10 }
```

Consider the call to **set** inside the **inc** method in **Counter** :

- In **late binding**, this call is attached to the concrete type.  
For an object of type **CounterExt**, the **set** method of **CounterExt** is used.

# Summary on the overloading polymorphism

- Overloading polymorphism allows the definition of values sharing multiply different implementations :

$$m ::= \{m_1 : T_1, m_2 : T_2, \dots, m_n : T_n\}$$

The types  $T_i$  may not share a common structure.

- Overloaded types may be modeled as finite **intersection types**.

$$T_1 \& T_2 \& \dots \& T_n = \bigcap_{i=1}^n T_i$$

- In order to use this polymorphism, a **selection** of the correct implementation for  $m$  is necessary, be it static or dynamic.
- The **late binding** is an example of dynamic selection in the case of overloaded types. It appears naturally in object-oriented programming.



- 1 Simple lambda-calculus
- 2 Polymorphism
- 3 Subtyping
  - Variance
  - The contravariance curse
  - Principles for variance
- 4 Proofs with types

# The subtyping test (1)

## Rationale

- A value can have multiple types, and the types share inclusion relations.
- By definition, a value  $v : T$  can also be typed  $v : T'$  whenever  $T \leq T'$ .

Consequence :

## ▶ Substitution Lemma

$T \leq T'$  iff every value  $v : T$  can be used in a context where  $T'$  is expected.

## Subtyping test (example in Java)

Attempt to assign a value of type  $T$  into a variable of type  $T'$  unchanged.

```
Integer one      = 1;  
Number super_one = one; // OK
```

```
Number pi      = 3.14;  
Double sub_pi = pi; // Type error
```

# The subtyping test (2)

## Problem : implicit conversions

As a subtyping test, it has false positives, because many languages allow implicit conversions of values.

## Example in C

Initializing an int variable with a float value forces an implicit conversion :

```
int main(void) {  
    int z = 3.14; // !!  
    z += 2.92;    // !!  
    printf("%d\n", z); }
```

```
% clang implicit.cpp -Wall -Wextra  
implicit.cpp:4:11: warning: implicit conversion  
from 'double' to 'int' changes value from 3.14 to 3  
    int z = 3.14;  
           ~   ^~~~  
1 warning generated.
```

- The subtyping relation is supposed to be **antisymmetric**, but the implicit conversions blur this property.
- In this course, we consider subtyping without implicit conversions.

# Summary on the different kinds of polymorphisms

- Each form of polymorphism defines new families of types, and these families share **subtyping relations**.
- The more families of types exist, the more **complex** the subtyping relation between types becomes.
- In this section, we explore some characteristics of this relation, and examine some implications in terms of programming.

Recall the subtyping theorems established for each form of polymorphism :

▶ **Lemma (Parametric subtyping) :**

Let  $\forall T_1..T_n, U$  be a parametric type.

- **Parametric subtyping** :  $\forall T_1..T_n, U <: [T_1 \mapsto U_1, .. T_n \mapsto U_n]U$ .

▶ **Lemma (Record subtyping) :**

Let  $T = \{l_i : T_i\}$  and  $T' = \{m_i : T'_i\}$  be two record types.

- **Width subtyping** : if  $T \supset T'$  as sets of pairs labels/types, then  $T <: T'$ .
- **Depth subtyping** : if  $T$  and  $T'$  share exactly the same labels and  $\forall i, T_i <: T'_i$ , then  $T <: T'$ .

▶ **Lemma (Intersection subtyping) :**

If  $T_1$  and  $T_2$  are two different types,  $T_1 \& T_2 <: T_1$  and  $T_1 \& T_2 <: T_2$

## Where do subtyping relations come from ?

- Some relations are **structural**, meaning that they are related to the form of the underlying type families.
- Some relations may be deduced as **consequences** of existing subtyping relations.

Given for instance :

- the `List[·]` type as a type function  $T \rightarrow \text{List}[T]$ ,
- two types `Banana` and `Fruit` such that `Banana <: Fruit`.

Can we deduce a subtyping relation between `List[Banana]` and `List[Fruit]` ?

For example :  $\text{List}[\text{Banana}] \overset{?}{<} \text{List}[\text{Fruit}]$

⇒ In other words, is the `List[·]` function increasing on types ?

# Variance

## Definition (Covariance / Contravariance / Invariance)

Let  $f : \text{Typ} \rightarrow \text{Typ}$  be a function on types.

- $f$  is said to be **covariant** iff it is increasing with regard to  $<:$

$$\forall T, U, \quad T <: U \Rightarrow f(T) <: f(U)$$

- $f$  is said to be **contravariant** iff it is decreasing with regard to  $<:$

$$\forall T, U, \quad T <: U \Rightarrow f(T) >: f(U)$$

- If the images  $f(T)$  and  $f(U)$  are always incomparable when  $T \neq U$ ,  $f$  is said to be **invariant**.

## Example

The type of the immutable lists `List[·]` can be considered to be **covariant**.

- The Java generics are invariant, the variance appearing in wildcards.

```
// "? extends Number" refers to any subtype of Number  
ArrayList<? extends Number> a = new ArrayList<Integer>();
```

- C# support variance for generic interfaces, but the classes are invariant.

```
public interface IEnumerable<out T> { // "out T" indicates the covariance  
    public IEnumerable<T> Append<T> (IEnumerable<T> source, T elem); }
```

- Scala supports variance for generic interfaces and classes.


```
class List[+A] { // "+A" indicates the covariance  
    def append[B >: A](x : B) : List[B] } // append is called "::" in Scala
```



# Function subtyping

## Variance of the function type

The type  $T \rightarrow U$  can be considered as a function (on types) of  $T$  and  $U$ . What kind of variance relations does it induce?

Recall  that subtyping can be thought of in terms of **substitution**.

Consider the following record type for storing  $\text{Nat}$  values :

$$S ::= \{\text{get} : \text{Unit} \rightarrow \text{Nat}, \text{set} : \text{Nat} \rightarrow \text{Unit}\}$$


Could such a store be replaced with one of the following types?

- $\{\text{get} : \text{Unit} \rightarrow \text{Int}, \text{set} : ..\}$  where  $\text{Int} <: \text{Nat}$
- $\{\text{get} : \text{Unit} \rightarrow \text{Number}, \text{set} : ..\}$  where  $\text{Nat} <: \text{Number}$

# Function subtyping

## Variance of the function type

The type  $T \rightarrow U$  can be considered as a function (on types) of  $T$  and  $U$ . What kind of variance relations does it induce?

Recall  that subtyping can be thought of in terms of **substitution**.

Consider the following record type for storing  $\text{Nat}$  values :

$$S ::= \{\text{get} : \text{Unit} \rightarrow \text{Nat}, \text{set} : \text{Nat} \rightarrow \text{Unit}\}$$


Could such a store be replaced with one of the following types?

- $\{\text{get} : \text{Unit} \rightarrow \text{Int}, \text{set} : ..\}$  where  $\text{Int} <: \text{Nat}$  
- $\{\text{get} : \text{Unit} \rightarrow \text{Number}, \text{set} : ..\}$  where  $\text{Nat} <: \text{Number}$  

# Function subtyping

## Variance of the function type

The type  $T \rightarrow U$  can be considered as a function (on types) of  $T$  and  $U$ . What kind of variance relations does it induce?

Recall  that subtyping can be thought of in terms of **substitution**.

Consider the following record type for storing  $\text{Nat}$  values :

$$S ::= \{\text{get} : \text{Unit} \rightarrow \text{Nat}, \text{set} : \text{Nat} \rightarrow \text{Unit}\}$$


Could such a store be replaced with one of the following types?

- $\{\text{get} : \dots, \text{set} : \text{Int} \rightarrow \text{Unit}\}$  where  $\text{Int} <: \text{Nat}$
- $\{\text{get} : \dots, \text{set} : \text{Number} \rightarrow \text{Unit}\}$  where  $\text{Nat} <: \text{Number}$

# Function subtyping

## Variance of the function type

The type  $T \rightarrow U$  can be considered as a function (on types) of  $T$  and  $U$ . What kind of variance relations does it induce?

Recall  that subtyping can be thought of in terms of **substitution**.

Consider the following record type for storing  $\text{Nat}$  values :

$$S ::= \{\text{get} : \text{Unit} \rightarrow \text{Nat}, \text{set} : \text{Nat} \rightarrow \text{Unit}\}$$

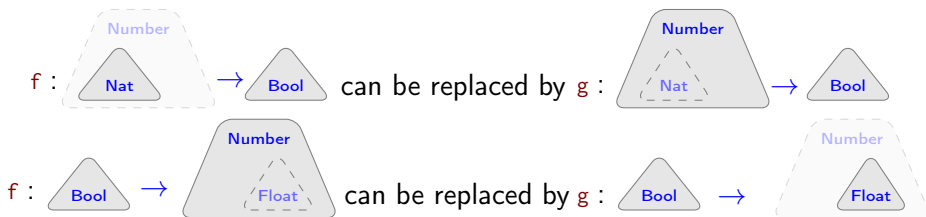
Could such a store be replaced with one of the following types?

- $\{\text{get} : \dots, \text{set} : \text{Int} \rightarrow \text{Unit}\}$  where  $\text{Int} <: \text{Nat}$  
- $\{\text{get} : \dots, \text{set} : \text{Number} \rightarrow \text{Unit}\}$  where  $\text{Nat} <: \text{Number}$  

## Lemma (Function subtyping) :

A function type is **covariant** in its **result type** and **contravariant** in its **parameter type**.

If  $T_{inf} <: T_{sup}$  and  $U_{inf} <: U_{sup}$ , then  $(T_{sup} \rightarrow U_{inf}) <: (T_{inf} \rightarrow U_{sup})$



# The contravariance curse

Subtyping is often used as a means to refine types (e.g. with inheritance). But the variance rules somewhat hinder these forms of refinements.

- Consider the example for a record type that compares numbers :

$$\text{EqNumber} ::= \{\text{val} : \text{Number}, \text{equal} : \text{Number} \rightarrow \text{Bool}\}$$

- Consider now inheriting from this with more precise internal numbers :

$$\text{EqFloat} ::= \{\text{val} : \text{Float}, \text{equal} : \text{Float} \rightarrow \text{Bool}\}$$

Deceptive (but also disappointing) fact

$$\text{EqFloat} \not\leq \text{EqNumber}$$

```

class Point {
    int x, y;
    Point(int _x, int _y)    { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                   (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other)    { return (other.c == c) &&
                                   (super.equals(other)); }
}

static boolean isOrigin(Point p) { return p.equal(new ColPoint(0,0,0)); }

```

- What happens when calling `isOrigin(new ColPoint(0,0,7))`?

```

class Point {
    int x, y;
    Point(int _x, int _y) { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                     (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other) { return (other.c == c) &&
                                           (super.equals(other)); }
}

static boolean isOrigin(Point p) { return p.equal(new ColPoint(0,0,0)); }

```

- What happens when calling `isOrigin(new ColPoint(0,0,7))` ?
  - `ColPoint.equal` ✗ `Point.equal` because of the contravariance rule.



```

class Point {
    int x, y;
    Point(int _x, int _y)    { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                     (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other)    { return (other.c == c) &&
                                     (super.equals(other)); }
}

static boolean isOrigin(Point p) { return p.equal(new ColPoint(0,0,0)); }

```

- What happens when calling `isOrigin(new ColPoint(0,0,7))` ?
  - `ColPoint.equal` ✗ `Point.equal` because of the contravariance rule.
  - `equal` is overloaded, it uses `Points` in `isOrigin` and returns `true`.

```

class Point {
    int x, y;
    Point(int _x, int _y)    { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                   (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other)    { return (other.c == c) &&
                                   (super.equals(other)); }
}

static boolean isOriginBis(Point p) { return p.equal(new Point(0,0)); }

```

- What happens if we suppose that `ColPoint.equal` <: `Point.equal` anyway?

```

class Point {
    int x, y;
    Point(int _x, int _y) { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                     (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other) { return (other.c == c) &&
                                           (super.equals(other)); }
}

static boolean isOriginBis(Point p) { return p.equal(new Point(0,0)); }

```

- What happens if we suppose that `ColPoint.equal` <: `Point.equal` anyway?
  - the call `isOriginBis(new ColPoint(0,0,7))` is well-typed,

```

class Point {
    int x, y;
    Point(int _x, int _y)    { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                   (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other)    { return (other.c == c) &&
                                   (super.equals(other)); }
}

static boolean isOriginBis(Point p) { return p.equal(new Point(0,0)); }

```

- What happens if we suppose that `ColPoint.equal` <: `Point.equal` anyway?
  - the call `isOriginBis(new ColPoint(0,0,7))` is well-typed,
  - it uses the `ColPoint` equality with a `Point` parameter, and gets stuck.

## Principles for variance of type variables

A type playing the role of a **supplier** can vary **covariantly** and must **not** vary contravariantly.

For example : r-values, getters, results of functions

A type playing the role of a **receiver** can vary **contravariantly** and must **not** vary covariantly.

For example : l-values, setters, parameters of functions

In Java, this takes the form of the “**Get and Put principle**” popularized by Naftalin and Wadler in *Java Generics* (2006) :

*“Use an **extends** wildcard when you only **get** values out of a structure, use a **super** wildcard when you only **put** values into a structure, and don't use a wildcard when you both get and put.”*

Consider two types `ColPoint <: Point`, and the following generic interfaces :

<code>GetF[T]</code>	<code>::= {get : Unit → T}</code>	covariant	<del>contravariant</del>
<code>SetF[T]</code>	<code>::= {set : T → Unit}</code>	contravariant	<del>covariant</del>

- The following examples are unsafe if the variance is reversed :

```
let gpt : GetF[Point] = ... in
(* Valid if GetF[.] is contravariant *)
let gcpt : GetF[ColPoint] = gpt in
(* Unsafe access to inexistant color field *)
gcpt.get().color
```

```
let gcpt : SetF[ColPoint] = ... in
(* Valid if SetF[.] is covariant *)
let gpt : SetF[Point] = gcpt in
(* Unsafe setting of missing color field *)
gpt.set(new Point())
```

- The following is an example of type unsafety in Java :

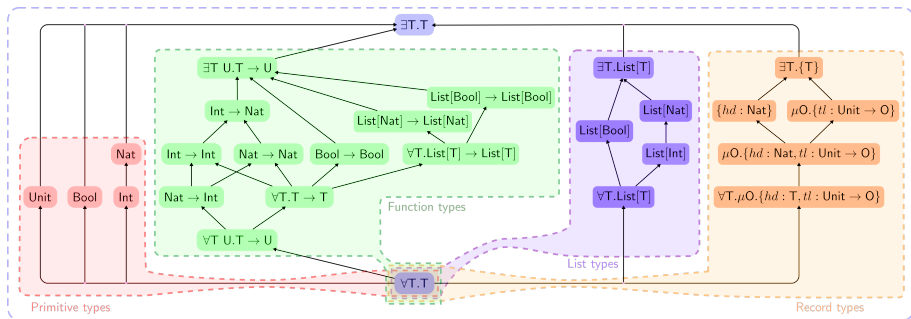
```
String[] strings = new String[1];
Object[] objects = strings;           // Arrays are covariant
objects[0] = new Integer(1);          // Runtime failure
```

The Java arrays implement both `GetF` and `SetF`, and should be invariant.

# Summary on subtyping

- Subtyping is an important property of a type system, enabling to use the polymorphism at full strength.
- The subtyping relations are refined by the notion of **variance**.
- Many constructions of the language (functions, records ...) are the source of specific subtyping rules.  
It is also possible to refine these relations by defining variance relations for particular type functions.
- Depending on the **coherence** of the relations, **type unsafety** problems may appear when programming.
- Deciding the subtyping relation also relies on verifying the coherence of the different local subtype relations.

# The type lattice





# Deciding the subtype relation (1)

In order to deal with object types, it is necessary to decide the subtyping relation on a wide family of types, among which recursive types.

## Subtyping decision algorithm

The function `subtype`( $\mathcal{A}$ ,  $S$ ,  $T$ ) decides whether  $S <: T$ .

- It considers a set of assumptions  $\mathcal{A}$ , each assumption being a pair  $(S_i, T_i)$  such that  $S_i <: T_i$ . Initially, the set is empty.
- Depending on the form of  $S$  and  $T$ , it deduces new assumptions.
- It terminates either when finding an incoherent set of assumptions or returns a set of coherent assumptions.

Basically, `subtype` builds a subset of the type lattice.

## Definition (Subtyping decision algorithm)

The function  $\text{subtype}(\mathcal{A}, S, T)$  is defined as :

- if  $S = T$  or  $(S, T) \in \mathcal{A}$ , return  $\mathcal{A}$
- if  $(T, S) \in \mathcal{A}$ , fail
- else let  $\mathcal{A}_0 = \mathcal{A} \cup (S, T)$  and depending on  $(S, T)$  :
  - $S = \{l_i : S_i\}_{i=[1..n+m]}$  and  $T = \{l_i : T_i\}_{i=[1..n]}$ , then compute in sequence  $\mathcal{A}_i = \text{subtype}(\mathcal{A}_{i-1}, S_i, T_i)$  for  $i \in [1..n]$ , return fail if any computation fails otherwise return  $\mathcal{A}_n$ .
  - $S = S_1 \rightarrow S_2$  and  $T = T_1 \rightarrow T_2$  then let  $\mathcal{A}_1 = \text{subtype}(\mathcal{A}_0, T_1, S_1)$  and  $\mathcal{A}_2 = \text{subtype}(\mathcal{A}_1, S_2, T_2)$  in return fail if any computation fails otherwise return  $\mathcal{A}_2$ .
  - $T = \mu X. T_1$  then compute  $\text{subtype}(\mathcal{A}_0, S, [X \mapsto \mu X. T_1] T_1)$
  - $S = \mu X. S_1$  then compute  $\text{subtype}(\mathcal{A}_0, [X \mapsto \mu X. S_1] S_1, T)$
  - otherwise fail.

## Deciding the subtype relation (2)

Consider the following two object types :

$$\begin{aligned} O &::= \mu X. \{\text{clone} : \text{Unit} \rightarrow X\} \\ S &::= \mu Y. \{\text{clone} : \text{Unit} \rightarrow Y, \text{val} : \text{Nat}\} \end{aligned}$$

The computation of  $\text{subtype}(S, O)$  passes through the following steps :

- $\text{subtype}(S, O)$
- $\text{subtype}(S, \{\text{clone} : \text{Unit} \rightarrow O\})$
- $\text{subtype}(\{\text{clone} : \text{Unit} \rightarrow S, \text{val} : \text{Nat}\}, \{\text{clone} : \text{Unit} \rightarrow O\})$
- $\text{subtype}(\text{Unit} \rightarrow S, \text{Unit} \rightarrow O)$
- $\text{subtype}(\text{Unit}, \text{Unit})$  and  $\text{subtype}(S, O)$  then terminates successfully.

As a result, this proves  $S <: O$ , even if they both are recursive types.

1 Simple lambda-calculus

2 Polymorphism

3 Subtyping

4 Proofs with types

- Phantom types
- Refinement types
- Dependent types

## General idea

Use types to enforce logic properties on the values they represent.


## Examples

- having values that can be compared (`Comparable`, `Eq a ...`)
- having numeric-like values (`Number`, `Num a ...`)
- having a list-like representation (`Cons`, `Nil`)

More generally, any form of interface can be seen as a logic property.

Can we generalize and find other sorts of properties represented by types?

# Representation of particular sets of values

Consider the following algebraic data type  for representing lists :

$$\text{List}[T] ::= \text{Nil} \quad | \quad \text{Cons}(T, \text{List}[T])$$

Suppose that this type is realized with the following constructors :

- $\text{nil} : \text{List}[T]$  is the empty list
- $\text{cons} : T \rightarrow \text{List}[T] \rightarrow \text{List}[T]$  is a constructor for lists.

The usual accessors  $\text{head}$  and  $\text{tail}$  are provided :

- $\text{head} : \text{List}[T] \rightarrow T$  returns the first element
- $\text{tail} : \text{List}[T] \rightarrow \text{List}[T]$  returns all but the first element.

These accessors are **problematic** : they are not defined for  $\text{nil}$ .  
Could these accessors be typed for only non-empty lists ?

# Refinement of sets of values

In this very case, it is natural to define two types :

- `EmptyList` containing the `nil` value
- `NonEmptyList[T]` containing the non-empty lists

Both types are naturally subtypes of `List[T]`.

With this refinement, the accessors can be defined as total functions :

- `head` : `NonEmptyList[T] → T` returns the first element
- `tail` : `NonEmptyList[T] → List[T]` returns all but the first element.

(`head_nil`) becomes a **non-typable** expression instead of stuck at runtime.

This idea can be generalized, restricting values to :

- non-zero or positive numeric values,
- open file descriptors in contrast to closed ones,
- non-null pointers in contrast to null ones.

It is desirable to have the possibility to refine the implementation and the logical properties **independently**.

- Otherwise, an implementation of an `AssocList[T] <: List[T]` must provide code for both empty and non-empty lists.
- One must devise a mechanism for attaching logical properties to existing types without hindering the usual inheritance mechanisms.



# Phantom types

## Definition (Phantom type)

A type  $T$  is said to be a **phantom type** if it has no influence at runtime, i.e its values never occur in any computation.

A type variable  $T$  in a parameterized type  $F[T]$  is said to be a **phantom type** if it is only meant to be applied to phantom types.

## Examples

```
interface Phantom {} // In Java
```

```
type phantom (* In OCaml *)
```

A sufficient condition to be a phantom type is to stand for the empty set of values.

## Example : write-restricted objects

Consider two phantom types `readonly` and `readwrite`.

Let us create a parameterized type `Readable[T]` such that `T` constrains its capabilities : only `Readable[readwrite]` can be modified.

```
(* Interface *)
module type REF = sig
  type 'a t
  val create : int → readwrite t
  val set    : readwrite t → int → unit
  val get    : 'a t → int
  val freeze : 'a t → readonly t
end
```

```
(* Implementation *)
module Ref : REF = struct
  type 'a t = int ref
  let create x = ref x
  let set r x = r := x
  let get r = !r
  let freeze x = x
end
```

```
let rw = create 4;;
let ro = freeze rw;;
set ro 7;; (* Type error : This expression has type readonly t
            (* but an expression was expected of type readwrite t *)
```

## Example : write-restricted objects


Consider two phantom types `readonly` and `readwrite`.

Let us create a parameterized type `Readable[T]` such that `T` constrains its capabilities : only `Readable[readwrite]` can be modified.

```
class Readable<A extends Access> {  
  int val;  
  Readable(int t){ val = t; }  
  
  static Readable<ReadWrite> create(int t)           { .. }  
  static void               set(Readable<ReadWrite> c, int t) { .. }  
  static int                get(Readable<?> c)         { .. }  
  static Readable<ReadOnly> freeze(Readable<?> c)     { .. }  
}
```

```
Readable<ReadWrite> rw = Readable.create(5);  
Readable<ReadOnly>  ro = Readable.freeze(rw);  
Readable.set(ro, 11); // Incompatible types: Readable<ReadOnly> cannot  
                     //      be converted to Readable<ReadWrite>
```

## Definition (GADT)


A **generalized algebraic datatype** is an algebraic datatype  containing a phantom type, and whose constructors can enforce restrictions on the phantom type.

## Example

```
type _ data =  
  | Int  : int          →      int data  
  | Str  : string       →      string data  
  | Pair : 'a data * 'b data → ('a * 'b) data
```

```
let x = Int 1 and y = Str "one" in Pair(x, y);; (* → (int*string) data *)
```

## Definition (GADT)

A **generalized algebraic datatype** is an algebraic datatype  containing a phantom type, and whose constructors can enforce restrictions on the phantom type.

## Example

```
type _ data =  
  | Int  : int          →      int data  
  | Str  : string       →      string data  
  | Pair : 'a data * 'b data → ('a * 'b) data
```

```
let add (Int u) (Int v) = Int(u+v);; (* int data → int data → int data *)
```

## Example : typed evaluator

In this example, a GADT is used to represent typed computations :

```
type _ expr =  
  | Bool : bool → bool expr  
  | Int  : int  → int  expr  
  | If   : bool expr * 'a expr * 'a expr → 'a expr  
  | Eq   : 'a expr * 'a expr → bool expr  
  | Add  : int  expr * int  expr → int  expr
```

The `eval` function returns the value encapsulated inside the expression :

```
let rec eval : type a. a expr → a = function (* with type  $\forall T, (\text{Expr}[T] \rightarrow T)$  *)  
  | Bool b      → b  
  | Int i       → i  
  | If (b, l, r) → if eval b then eval l else eval r  
  | Eq (a, b)   → (eval a) = (eval b)  
  | Add (a,b)   → (eval a) + (eval b) ;; (* Addition on integers *)
```

- A GADT value is an existential value, involving runtime checking.
- The compiler checks the constraints for each constructor individually.

# Reification of types

- Once phantom types have been attached to other types, it becomes natural to apply computations on these.

<code>plus(int, float) <math>\Rightarrow</math> float</code>	<code>append(Int, String) <math>\Rightarrow</math> Vector&lt;Any&gt;</code>
<code>plus(int, int) <math>\Rightarrow</math> int</code>	<code>append(Char, String) <math>\Rightarrow</math> String</code>

- Not all languages allow computations at the type level, and therefore mimic these computations at the value level.

## Definition (Reification)

A set of types  $\mathcal{T} ::= \{T_i\}$  is said to be **reified** into a set of values  $\mathcal{V} ::= \{v_i\}$  if there exists a bijection between the  $\mathcal{T}$  and  $\mathcal{V}$ .

Ideally, the set  $\mathcal{V}$  is represented as (another) type supporting this bijection. If both sets are of size 1, the type is called a **singleton type**.

# Example : reification of naturals

In this example, the phantom types represent the **Peano naturals** :

```
type zero      (* Type for representing zero *)
type 'a succ   (* Type for representing the successor *)

type _ nat =   (* Bridge Nat[T] between values and types *)
| NZ : zero nat      (* Value for representing zero *)
| NS : 'a nat → ('a succ) nat  (* Value for representing the successor *)
```

- `NZ` and `NS` are values typed by `Nat[T]` in bijection with the naturals :

$$\underbrace{(\text{NS } \dots \text{ NS } \text{NZ})}_{k \text{ times}} : \underbrace{(\text{succ } \dots \text{ succ } \text{zero})}_{k \text{ times}}$$

- Computations on types can be carried over onto values :

```
let rec nat_to_int : type a. a nat → int = fun x → match x with
| NZ   → 0
| NS n → 1 + nat_to_int n
```



## Example : length-encoded lists

Let's extend this example to create lists whose type contains their length :

```
type (_,_) seq = (* 1st parameter = type of elements, 2nd parameter = length *)  
  | Nil   : ('a, zero) seq  
  | Cons : 'a * ('a, 'n) seq → ('a, 'n succ) seq
```

```
let rec head : type a n. (a, n succ) seq → a = function  
  | Cons(x, _) → x
```

```
let rec tail : type a n. (a, n succ) seq → (a, n) seq = function  
  | Cons(_, s) → s
```

```
let rec map : type a b n. (a → b) → (a,n) seq → (b,n) seq =  
  fun f l → match l with  
    | Nil       → Nil  
    | Cons (x, s) → Cons (f x, map f s)
```

- The type of `map` encodes the fact that it preserves the length of lists.

## Example : length-encoded lists

- In Haskell, it's even possible to express computations on types :

```
data Zero          -- Phantom types for naturals
data Succ nat

type family nat1 :+: nat2 :: * -- Type family for the “:+” function on naturals

type instance Zero      :+: nat2 = nat2
type instance Succ nat1 :+: nat2 = Succ (nat1 :+: nat2)
```

- This yields the following type for the concatenation on lists :

```
(++) :: List a len1 → List a len2 → List a (len1 :+: len2)
Nil      ++ list = list
Cons el els ++ list = Cons el (els ++ list)
```

- The type of `++` encodes the fact that the length of the concatenation of two lists is the sum of the lengths of its components.

# Composition of static properties

## Problem

Annotated types do not compose well in general.

Computations  
returning an integer

Computations  
applying a division

1) Consider the example of the `mean` function on lists of integers :

```
mean ::= fun l → let n = List.length l in (sum l) / n
```

- The `length` function **must** return a generic non-negative integer.
- The division function **should** take a generic positive integer.

# Composition of static properties

## Problem

Annotated types do not compose well in general.

Computations re-  
turning a list

Computations taking  
the **head** of a list

2) Consider a function returning the first even integer in a list :

```
fst_even ::= fun l → let m = filter is_even l in head m
```

Without static knowledge that **m** is non-empty, one must check dynamically.

# Composition of static properties

## Problem

Annotated types do not compose well in general.

Computations re-  
turning a list

Computations taking  
the **head** of a list

2) Consider a function returning the first even integer in a list :

```
fst_even ::= fun l → let m = filter is_even l in head m
```

Without static knowledge that **m** is non-empty, one must check dynamically.

This is a consequence of the **undecidability of evaluation** : logic properties that evolve at runtime cannot be decided statically in general.

# Strategy for composing properties

In some cases, it is possible to provide a **static** proof of the property.

Consider the problem of accessing the  $n$ -th element of a list :

`get : Nat → List[T] → T`

How could we make `get` access only concrete indices of the list ?

- Construct a type `Leq[m,n]` expressing the fact that  $m \in [0; n[$  :

```
data Leq[m,n] where
  LessZ :: Leq[Zero,Succ n]           -- Proof that :
  LessS :: Leq[m,n] → Leq[Succ m,Succ n] -- 0 < n
                                           -- if m < n, then m+1 < n+1
```

- A value of type `Leq[m,n]` is computed dynamically when required.

## Example : lists with safe access

- The GADT reifying the property  $m < n$  :

```
data Leq[m,n] where
  LessZ :: Leq[Zero,Succ n]
  LessS :: Leq[m,n] → Leq[Succ m,Succ n]
```

- The `less-than` function computes a proof that  $m < n$  (if any) :

```
lt :: Nat m → Nat n → Maybe (Leq[m,n])
lt Zero      (Succ n) = Just LessZ
lt (Succ m) (Succ n) = case lt m n of Some proof → Some (LessS proof)
                                Nothing      → Nothing
lt _         _       = Nothing
```

- The type-safe `get` function can only access safe indices of a list :

```
get :: Leq[m,n] → List[a,n] → a
get LessZ      (Cons x xs) = x
get (LessS k) (Cons x xs) = get k xs
```

# Refinement types

Going further along these lines, it is possible to attach a proof-checker to help the compilation phase, as is done in Liquid Haskell or in Dafny.

Consider the problem of defining a type-safe `divide` function on integers :

```
type NonZero = { v : Int | v /= 0 } -- type for non-zero integers

divide :: Int → NonZero → Int
divide _ 0 = die "divide_by_zero"    -- can never happen
divide n d = n `div` d
```

- A type attached with a logical property is called a **refinement type**.
- Logical assertions are transferred and checked by a SMT solver.



## Example : iterating on vectors (1)

Here, `loop` iterates a function over the integers in the interval `[lo;hi[` :

```
loop :: lo:Nat → hi:{Nat|lo <= hi} → a → (Btwn lo hi → a → a) → a
loop lo hi base f = loop_rec base lo where
  loop_rec acc i | i < hi    = loop_rec (f i acc) (i + 1)
  loop_rec _   _ | otherwise = acc
```

Typically, `loop 0 n x0 f` computes the sequence :

$$\begin{cases} x_0 \text{ given} \\ x_{k+1} = f(k, x_k) \end{cases}$$

The type of the `loop` function is verified by the compiler and ensures that :

- `lo ≤ hi`, forming an interval `Btwn lo hi ::= [lo;hi[`;
- `f` accesses only integers in the interval `Btwn lo hi`.

## Example : iterating on vectors (2)

The `loop` function can then be used to write a `dotProduct` function :

```
loop :: lo:Nat → hi:{Nat|lo <= hi} → a → (Btwn lo hi → a → a) → a
```

```
dotProduct :: x:[Int] → { y:[Int] | len x = len y } → Int
dotProduct x y = loop 0 n 0 body where
  n           = length x
  body i acc = acc + (x ! i) * (y ! i)
```

- The compiler is able to infer that the indices accessed are always valid.
- This function **only** requires a proof that both vectors have same length.
- It does **not** need to check that all the array accesses are safe.

# Termination proofs

Liquid Haskell is able to prove the **termination** of the following function :

```
fib :: i:Int → Int
fib i | i == 0    = 0
      | i == 1    = 1
      | otherwise = fib (i-1) + fib (i-2)
```

- Applying a series of well-chosen heuristics, the compiler finds a well founded metric that decreases at each recursive call.
- More generally, it can automatically prove termination for a particular but **expressive** class of recursive functions (▶ strong normalization).

... which in itself is a pretty amazing feat.

# The frontier of automaticity

In some cases, the compiler is not able to infer the proofs **automatically**.

- More complex calculi exist with particularly powerful type systems.

Examples : Martin-Löf's type theory, the calculus of constructions ...

- As type inference became undecidable for  $\lambda_2$ , it is not surprising that it remains **undecidable** for more powerful calculi.

These proofs may be provided, possibly with the help of a proof-assistant.

- Proofs become another software component, at the same level as code.

Examples : languages with proof assistants such as Coq, Agda, Idris, ...

## Definition (Dependent type)

A **dependent type** is a type whose definition is parameterized by a value.

Note : allowing values inside types dramatically complexifies a type system

### Example

The type  $\text{Vec}[n, A]$  of the vectors of  $n$  elements of type  $A$ .  
It is technically called a dependent product written  $\prod_{n \in \mathbb{N}} \text{Vec}_n[A]$ .

**Inductive**  $\text{vec } a : \text{nat} \rightarrow \text{Type} := (* \text{Dependent type written as a function} *)$   
|  $\text{nil} : \text{vec } a \ 0$   
|  $\text{cons} : \text{forall } (h:a) (n:\text{nat}), \text{vec } a \ n \rightarrow \text{vec } a \ (S \ n).$

**Definition**  $\text{hd} \quad \{a\} \{n\} \quad (v:\text{vec } a \ (S \ n)) \quad : a$

**Definition**  $\text{tl} \quad \{a\} \{n\} \quad (v:\text{vec } a \ (S \ n)) \quad : \text{vec } a \ n$

**Definition**  $\text{nth} \quad \{a\} \{n\} \{p\} \quad (v:\text{vec } a \ n) \ (H: p < n) \quad : a$

**Fixpoint**  $\text{append} \quad \{a\} \{n\} \{p\} \quad (v:\text{vec } a \ n) \ (w:\text{vec } a \ p) : \text{vec } a \ (n+p)$

# Proof example : associativity of concatenation

```
data List a = Nil | a ::: List a deriving (Eq)
-- Definition of a concatenation function '++' on lists
Nil      ++ ys = ys
(x ::: xs) ++ ys = x ::: (xs ++ ys)
```

```
assocThm xs ys zs = (xs ++ ys) ++ zs == xs ++ (ys ++ zs)
```

```
assocPf :: xs:_ → ys:_ → zs:_ → { assocThm xs ys zs }
```

```
assocPf Nil ys zs = (Nil ++ ys) ++ zs
                  ==.   ys ++ zs
                  ==. Nil ++ (ys ++ zs)
```

```
assocPf (x ::: xs) ys zs = ((x ::: xs) ++ ys) ++ zs
                          ==. (x ::: (xs ++ ys)) ++ zs
                          ==. x ::: ((xs ++ ys) ++ zs)
                          ==. x ::: (xs ++ (ys ++ zs)) ? assocPf xs ys zs
                          ==. (x ::: xs) ++ (ys ++ zs)
```

# Conclusion

- Type systems offer a general framework to verify the safety of the composition of programming expressions.
- The association between **types** and **logic properties** is natural in this framework and mechanisms exist to facilitate this association :



- These logic properties constitute another form of programming. Types / proofs become a natural component accompanying the code.
- The mechanisms for the **verification** of these properties **grow in complexity** accordingly with the expressivity of the properties : type annotations, SMT-solvers ... up to proof assistants.
- **Undecidability** problems occur for the highest levels of complexity, hindering the verification capabilities for programmers.

# The present and future of type systems

- The development of more recent type systems and even more recent programming languages displays a high level of activity.
- As an example, regions constitute a mechanism to describe zones of code and memory determined statically.
- Effects systems restrict the kind of operations allowed in certain of these regions, typically reading or writing to memory.

Many of these languages are experimentations derived from **Haskell**.

- The **Rust** programming language is an example of the last generation of general-purpose languages incorporating some of these advances.
- It claims solving the problems of dangling pointers, uses-after-free and even data races for some classes of concurrent programs.