

Type Systems and Programming

D. Renault

ENSEIRB-Matmeca

Feb. 5th 2025, v.1.5.1

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

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**.

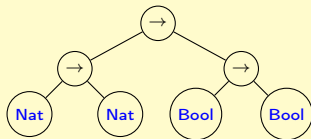
Definition (Types)

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** Γ is a possibly empty sequence of typings.

Definition (Typing deduction)

To **deduce a typing** from Γ , noted $\Gamma \vdash t : T$, consists in building a derivation tree using Γ 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 Γ :

- 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 $t_{app} ::= (t_1 \cdot t_2)$ in an environment Γ .

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

Example : application and abstraction

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

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

Example : application and abstraction

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

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

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

Example : application and abstraction

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

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

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

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

Example : application and abstraction

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

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

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

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

Example : application and abstraction

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

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

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

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

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

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

Example : derivation tree of a typing

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

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

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

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 λ -calculus** or λ_{\rightarrow} is defined as the set of the typable λ -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 λ -calculus** or λ_{\rightarrow} is defined as the set of the typable λ -expressions in the **Typ** family of types with the following typing rules :

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

Comparison with the rules in propositional logic :

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

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

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

Properties of the simply typed λ -calculus (1)

Theorem (Strong normalization) :

In λ_{\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 λ_{\rightarrow} extended with **recursion** and a type for naturals is a Turing-complete language.

Properties of the simply typed λ -calculus (2)

The type system of λ_{\rightarrow} is coherent with regard to β -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 β -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 λ -expressions that are not typable in λ_{\rightarrow} .

Example

The expression $nt ::= \lambda x.(x\ x)$ is not typable in λ_{\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 λ_{\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 λ_{\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** μ 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

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

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

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

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

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

$T ::=$...
 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

$T ::=$...
 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

$T ::=$...
 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

$T ::=$...
 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	<i>(* Nil is a constant *)</i>
Cons(2, Nil)	<i>(* Cons takes 2 arguments *)</i>
Cons(1, Cons(2, Cons(3, Nil)))	<i>(* their composition yields complex lists *)</i>

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

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

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** σ 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 substutions should be handled carefully.

Definition (Type constraints)

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

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

Definition (Unification)

The substitution σ is said to **unify** \mathcal{C} iff for all equation $S = T$ in \mathcal{C} , σS and σT 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 : Bool used where 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 Γ , 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

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

Example : deduction of a typing

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

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 σ that unifies all the constraints.

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

- σ 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 λ_{\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 λ -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

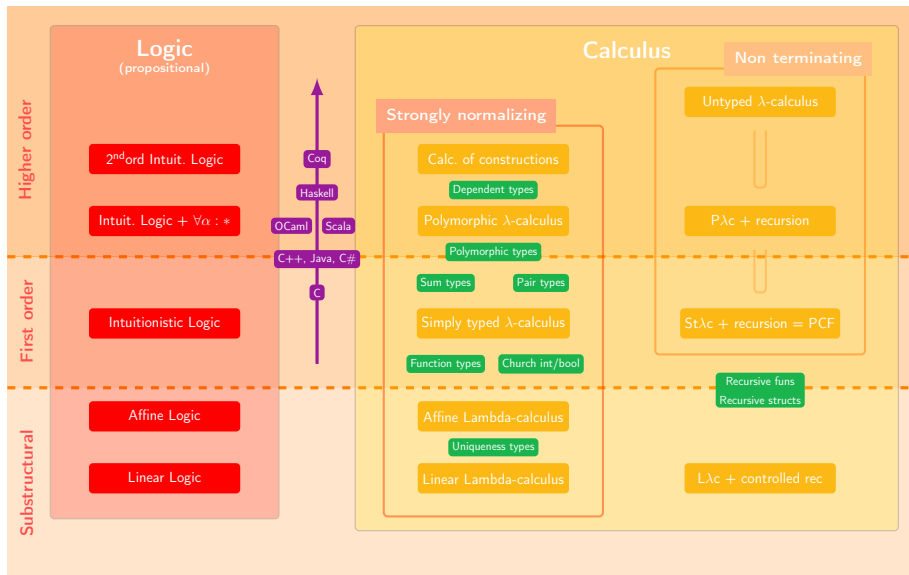
General tactics

Associate a typed λ -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 λ -calculus

Up until now, the framework we developed around λ_{\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.

- 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.