Type Systems and Programming

D. Renault

ENSEIRB-Matmeca

Jan. 15th 2025, v.1.5.1

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

Introduction

What's a programming language ?

```c
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));
}
```

```apl
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**.

Introduction

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 ?

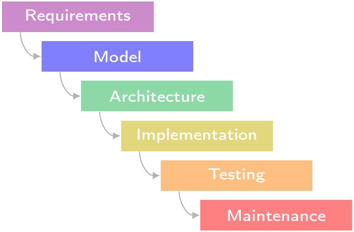
And more pragmatically, checking for the presence or absence of :

- 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 ?

$\Rightarrow$ Different families of methods, spread along the development cycle.



- Formal verification
  - Deductive methods
  - Model checking . . .
- Static analysis
  - Lexical analysis
  - Type systems . . .
- Runtime verification
  - Testing
  - Monitoring . . .

$\Rightarrow$ 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`
- Haskell (ghc-9.4) `haskell.org/ghc`
- LiquidHaskell (0.9.4-git) `ucsd-progsys.github.io/liquidhaskell-blog`
- Scala (2.13) `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.

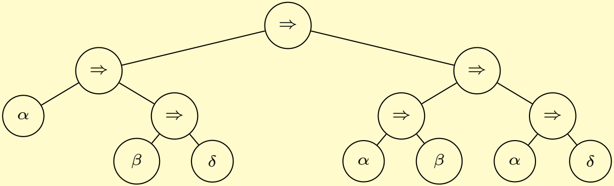
Overview

1. **Simple lambda-calculus**
   - Propositional logic
   - Untyped lambda calculus

Definition (Minimal intuitionistic logic)

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

- an infinite set of atomic formulae denoted as variables $\alpha, \beta, \ldots,$
- 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 :

- Γ is a possibly empty sequence of formulae called the **antecedents**,
- and α is a formula called the **consequent**.

Writing $\Gamma, P \vdash Q$ means that the antecedents are constituted of a list of formulae Γ 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 :

$$\frac{}{P \vdash P}\,[\text{ax}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\,[\Rightarrow\text{i}] \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q}\,[\Rightarrow\text{e}]$$

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

$$\Big(R \Rightarrow (S \Rightarrow T)\Big) \Rightarrow \Big((R \Rightarrow S) \Rightarrow (R \Rightarrow T)\Big)$$

Inference rules

$$\frac{}{P \vdash P}\,[\mathrm{ax}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\,[\Rightarrow\mathrm{i}] \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q}\,[\Rightarrow\mathrm{e}]$$

Proof as a derivation tree

$$\vdash (R \Rightarrow (S \Rightarrow T)) \Rightarrow ((R \Rightarrow S) \Rightarrow (R \Rightarrow T))$$

Frege's theorem

$$\Big(R \Rightarrow (S \Rightarrow T)\Big) \Rightarrow \Big((R \Rightarrow S) \Rightarrow (R \Rightarrow T)\Big)$$

Inference rules

$$\frac{}{P \vdash P}\,[\text{ax}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\,[\Rightarrow\text{i}] \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q}\,[\Rightarrow\text{e}]$$

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

$$\Big(R \Rightarrow (S \Rightarrow T)\Big) \Rightarrow \Big((R \Rightarrow S) \Rightarrow (R \Rightarrow T)\Big)$$

Inference rules

$$\frac{}{P \vdash P}\,[\mathrm{ax}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\,[\Rightarrow\mathrm{i}] \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q}\,[\Rightarrow\mathrm{e}]$$

Proof as a derivation tree

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

Frege's theorem

$$\Big(R \Rightarrow (S \Rightarrow T)\Big) \Rightarrow \Big((R \Rightarrow S) \Rightarrow (R \Rightarrow T)\Big)$$

Inference rules

$$\frac{}{P \vdash P}\,[\text{ax}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\,[\Rightarrow\text{i}] \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q}\,[\Rightarrow\text{e}]$$

Proof as a derivation tree

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

Frege's theorem

$$\Big(R \Rightarrow (S \Rightarrow T)\Big) \Rightarrow \Big((R \Rightarrow S) \Rightarrow (R \Rightarrow T)\Big)$$

Inference rules

$$\frac{}{P \vdash P}\,[\text{ax}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\,[\Rightarrow\text{i}] \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q}\,[\Rightarrow\text{e}]$$

Proof as a derivation tree

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

Frege's theorem

$$\Big(R \Rightarrow (S \Rightarrow T)\Big) \Rightarrow \Big((R \Rightarrow S) \Rightarrow (R \Rightarrow T)\Big)$$

Inference rules

$$\dfrac{}{P \vdash P}\,[\text{ax}] \qquad \dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\,[\Rightarrow\text{i}] \qquad \dfrac{\Gamma \vdash P \qquad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q}\,[\Rightarrow\text{e}]$$

Proof as a derivation tree

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

Frege's theorem

$$\Big(R \Rightarrow (S \Rightarrow T)\Big) \Rightarrow \Big((R \Rightarrow S) \Rightarrow (R \Rightarrow T)\Big)$$

Inference rules

$$\frac{}{P \vdash P}\,[\text{ax}] \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\,[\Rightarrow\text{i}] \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q}\,[\Rightarrow\text{e}]$$

Proof as a derivation tree

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

Frege's theorem

$$\Big(R \Rightarrow (S \Rightarrow T)\Big) \Rightarrow \Big((R \Rightarrow S) \Rightarrow (R \Rightarrow T)\Big)$$

Inference rules

$$\dfrac{}{P \vdash P}\,[\text{ax}] \qquad \dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\,[\Rightarrow\text{i}] \qquad \dfrac{\Gamma \vdash P \qquad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q}\,[\Rightarrow\text{e}]$$

Proof as a derivation tree

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

✓

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 λ-calculus**.

So let's start again . . .

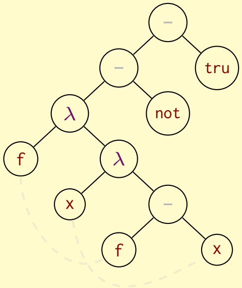
Definition (Untyped λ-calculus)

The untyped λ-calculus is the set of expressions t, u, … constructed from :

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

Example :

$$((\lambda f.\lambda x.(f␣x)␣not)␣true)$$



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 λ-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 :

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

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 (α-conversion)

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

$\alpha\mathrm{cnv}_{\mathtt{x}\to\mathtt{y}}(\mathtt{t})$ is defined as :

- $\alpha\mathrm{cnv}_{\mathtt{x}\to\mathtt{y}}(\lambda\mathtt{x.u}) = \lambda\mathtt{y}.\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{v})$
- $\alpha\mathrm{cnv}_{\mathtt{x}\to\mathtt{y}}(\mathtt{t}) = \mathtt{t}$ otherwise

... where $\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{t})$ is defined as :

- $\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{z}) = \mathtt{y}$ if $\mathtt{z} = \mathtt{x}$,
  $= \mathtt{z}$ otherwise
- $\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\lambda\mathtt{z.w}) = \lambda\mathtt{z.w}$ if $\mathtt{z} = \mathtt{x}$,
  $= \lambda\mathtt{z}.\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{w})$ otherwise
- $\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{v}\,\_\,\mathtt{w}) = (\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{v})\,\_\,\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{w}))$

Examples

- $\alpha\mathrm{cnv}_{x\to y}(\lambda\mathtt{x.x}) = \lambda\mathtt{y.y}$
- $\alpha\mathrm{cnv}_{x\to y}(\lambda\mathtt{x}.((\lambda\mathtt{x.x})\,\_\,\mathtt{x})) = \lambda\mathtt{y}.((\lambda\mathtt{x.x})\,\_\,\mathtt{y})$

Definition (α-conversion)

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

$\alpha\mathrm{cnv}_{\mathtt{x}\to\mathtt{y}}(\mathtt{t})$ is defined as :

- $\alpha\mathrm{cnv}_{\mathtt{x}\to\mathtt{y}}(\lambda\mathtt{x.u}) = \lambda\mathtt{y}.\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{v})$
- $\alpha\mathrm{cnv}_{\mathtt{x}\to\mathtt{y}}(\mathtt{t}) = \mathtt{t}$ otherwise

. . . where $\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{t})$ is defined as :

- $\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{z}) = \mathtt{y}$ if $\mathtt{z} = \mathtt{x}$,
  $= \mathtt{z}$ otherwise
- $\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\lambda\mathtt{z.w}) = \lambda\mathtt{z.w}$ if $\mathtt{z} = \mathtt{x}$,
  $= \lambda\mathtt{z}.\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{w})$ otherwise
- $\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{v}\,\_\,\mathtt{w}) = (\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{v})\,\_\,\mathrm{r}_{\mathtt{x}\to\mathtt{y}}(\mathtt{w}))$

- **Barendregt convention** : give distinct names to distinct bound variables.
- The λ-expressions can be considered **equivalent** up to α-conversion.

Definition (Substitution)

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

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

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

Example

$$[\mathtt{x} \mapsto \mathtt{biiip}]\Big((\lambda\mathtt{z}.(\mathtt{x}\ \mathtt{z}))\ \mathtt{y}\Big) = (\lambda\mathtt{z}.(\mathtt{biiip}\ \mathtt{z}))\ \mathtt{y}$$

Definition (β-reduction)

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

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

- $(\lambda x.v)\_w \to_\beta [x \mapsto w]v$
- if $u \to_\beta v$ then

$$\left.\begin{array}{l} \left.\begin{array}{ll} \bullet\ \text{Function reduction :} & (u\_w) \to_\beta (v\_w) \\ \bullet\ \text{Parameter reduction :} & (w\_u) \to_\beta (w\_v) \end{array}\right\}\text{Weak red.} \\ \begin{array}{ll} \bullet\ \text{Body reduction :} & \lambda x.u \to_\beta \lambda x.v \end{array} \end{array}\right\}\text{Strong red.}$$

Example

$$\Big(\lambda z.(\lambda x.x+1)\_(z+2)\Big)\_(3+4) \to_\beta \ldots \to_\beta \big((3+4)+2\big)+1$$

Definition (β-reduction)

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

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

- $(\lambda \mathtt{x}.\mathtt{v})\_\mathtt{w} \rightarrow_\beta [\mathtt{x} \mapsto \mathtt{w}]\mathtt{v}$
- if $\mathtt{u} \rightarrow_\beta \mathtt{v}$ then
  - Function reduction : $(\mathtt{u}\_\mathtt{w}) \rightarrow_\beta (\mathtt{v}\_\mathtt{w})$
  - Parameter reduction : $(\mathtt{w}\_\mathtt{u}) \rightarrow_\beta (\mathtt{w}\_\mathtt{v})$
  - Body reduction : $\lambda \mathtt{x}.\mathtt{u} \rightarrow_\beta \lambda \mathtt{x}.\mathtt{v}$

Weak red.

Strong red.

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

Properties of the λ-calculus

The λ-calculus endowed with the β-reduction relation is a Turing-complete computational model.

- **Church-Rosser theorem** : the β-reduction relation is **confluent**.
- There exist λ-expressions for which the evaluation is **infinite**.

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

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

Undecidability is at the heart of dealing with programming languages.

Syntax			Evaluation rules
t ::=		*expressions*	$\dfrac{t_1 \rightarrow_\beta t_1'}{(t_1\ t_2) \rightarrow_\beta (t_1'\ t_2)}$
	x	*variable*	
	λx.t	*abstraction*	$\dfrac{t \rightarrow_\beta t'}{(v\ t) \rightarrow_\beta (v\ t')}$
	(t t)	*application*	
v ::=		*values*	$(\lambda x.t_1\ t_2) \rightarrow_\beta [x \mapsto t_2]t_1$
	λx.t	*abstraction value*	

- **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 :

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

(these are the classical projection functions)
The booleans can now be manipulated with the following expressions :

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

Example

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

Church encodings : booleans

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

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

(these are the classical projection functions)
The booleans can now be manipulated with the following expressions :

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

Example

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

Church encodings : booleans

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

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

(these are the classical projection functions)
The booleans can now be manipulated with the following expressions :

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

Example

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

Church encodings : booleans

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

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

(these are the classical projection functions)
The booleans can now be manipulated with the following expressions :

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

Example

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

Church encodings : naturals

And the following expressions encode the natural numbers :

`zero` ::= λ`f`.λ`x`.`x`	`f` applied zero times
`succ` ::= λ`i`.λ`f`.λ`x`.(`f` ␣ ((`i` ␣ `f`) ␣ `x`))	`f` applied once to the result of (`i` ␣ `f`)

With the addition and multiplication functions defined as follows :

`plus` ::= λ`i`.λ`j`.((`i` ␣ `succ`) ␣ ((`j` ␣ `succ`) ␣ `zero`))	apply `succ`, first `j` times then `i` times to `zero`
`mult` ::= λ`i`.λ`j`.((`j` ␣ (`plus` ␣ `i`)) ␣ `zero`)	apply (`plus` ␣ `i`), `j` times to `zero`

Example : derivation tree of a β-reduction

In the λ-calculus extended with the Church boolean values :

$$\texttt{(not\textvisiblespace true)} \rightarrow_\beta \dfrac{\dfrac{\texttt{(if\textvisiblespace true)} \quad \rightarrow_\beta \quad \lambda\texttt{t}.\lambda\texttt{e}.\texttt{((true\textvisiblespace t)\textvisiblespace e)}}{\texttt{((if\textvisiblespace true)\textvisiblespace false)} \rightarrow_\beta (\lambda\texttt{t}.\lambda\texttt{e}.\texttt{((true\textvisiblespace false)\textvisiblespace e)\textvisiblespace false)}} \rightarrow_\beta \lambda\texttt{e}.\texttt{((true\textvisiblespace false)\textvisiblespace e)}}{\texttt{(((if\textvisiblespace true)\textvisiblespace false)\textvisiblespace true)} \quad \rightarrow_\beta \quad (\lambda\texttt{e}.\texttt{((true\textvisiblespace false)\textvisiblespace e)\textvisiblespace true)}}$$

$$\cdots \rightarrow_\beta \dfrac{\texttt{(true\textvisiblespace false)} \quad \rightarrow_\beta \quad \lambda\texttt{y}.\texttt{false}}{\texttt{((true\textvisiblespace false)\textvisiblespace true)} \rightarrow_\beta (\lambda\texttt{y}.\texttt{false\textvisiblespace true)}} \rightarrow_\beta \texttt{false}$$

The untyped λ-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`.

$$(\mathtt{succ}\ \mathtt{true}) \rightarrow_\beta (\underbrace{\lambda \mathtt{i}\,\mathtt{f}\,\mathtt{x}.(\mathtt{f}\ ((\mathtt{i}\ \mathtt{f})\ \mathtt{x}))}_{\mathtt{succ}}\ \underbrace{(\lambda \mathtt{u}\,\mathtt{v}.\mathtt{u})}_{\mathtt{true}}) \rightarrow_\beta \underbrace{\lambda \mathtt{f}\,\mathtt{x}.(\mathtt{f}\ \mathtt{f})}_{??}$$

Let's do it anyway ...

Extension of the λ-calculus : booleans & naturals

Syntax

`t ::=`	`...`	*expressions*
	`true`, `false`	*booleans*
	`zero`, `succ t`	*naturals*
	`if t then t else t`	*if-then-else*
	`iszero t`	*zero-equality*
`v ::=`	`...`	*values*
	`true`, `false`	*boolean value*
	`nv`	*numeric value*
`nv ::=`		*numeric values*
	`zero`	*zero value*
	`succ nv`	*successor value*

Evaluation rules

$$\frac{\mathtt{t_1} \rightarrow_\beta \mathtt{t_1'}}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \rightarrow_\beta \mathtt{if\ t_1'\ then\ t_2\ else\ t_3}}$$

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

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

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

$$\mathtt{iszero\ zero} \rightarrow_\beta \mathtt{true}$$

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

Example : derivation tree of a β-reduction

Examples of evaluation in the λ-calculus with booleans and integers :

- $$\dfrac{\texttt{iszero (succ zero)} \rightarrow_\beta \texttt{false}}{\texttt{if (iszero (succ zero)) then zero else (succ zero)} \rightarrow_\beta \texttt{if false then zero else (succ zero)}} \rightarrow_\beta \texttt{succ zero}$$
- $$\dfrac{\texttt{if false then zero else false} \rightarrow_\beta \texttt{false}}{\texttt{succ (if false then zero else false)} \rightarrow_\beta \texttt{succ false}} \rightarrow_\beta \texttt{??}$$

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 λ-calculus

Starting from now, we consider that the booleans and naturals are part of the definition of the λ-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.

Bibliography

- 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.