Lambda-calculus and programming language semantics

Thibaut Balabonski @ UPSay
Winter 2023
https://www.lri.fr/~blsk/LambdaCalculus/

Chapter 1: lambda-calculus

1 A computational theory of function

Timeline

1870 Which ground for mathematics ? Sets or functions ?

1920 Moses Schönfinkel, Haskell Curry: combinatory logic. Basic blocks for building functions.

1936 Alonzo Church: λ-calculus. Characterization of computable functions. Equivalent to Turing machines. Solves the *Entscheidungsproblem.*

1970+ λ-calculus grows together with computer science. Functional programming. Proof assistants.

Functions

One concept, various notations.

Maths	$x \mapsto x$
	$f \mapsto (x \mapsto f(f(x)))$
Caml	`fun x -> x`
	`fun f -> (fun x -> f(f x))`
Python	`lambda x: x`
	`lambda f: (lambda x: f(f(x)))`
λ-calculus	$\lambda x.x$
	$\lambda f.(\lambda x.f(f\ x))$

2 λ-calcul: basic definitions

The λ-calculus is defined by a set of *terms*, which represent programs or algorithms, and by *conversion rules*, which describe how computation is performed.

Terms (expressions)

The λ-calculus syntax consists of a notion of *expression*, or *term.* Terms are built using three constructs.

x variable, reference to a function parameter

$t_1 t_2$ application of a term t_1 to a term t_2, t_1 is to be seen as a function and t_2 as its given argument.

$\lambda x.t$ function with a single parameter x, whose result is given by t

Functions are defined by their *behaviour.*

Examples

- Identity
$$\lambda x.x$$
takes a paremeter x and returns the value of x
- Constant functions generator
$$\lambda c.(\lambda x.c)$$
takes a parameter c and returns a constant function whose result is constantly c
- Distribution
$$\lambda x.(\lambda y.(\lambda z.((x\ z)\ (y\ z))))$$
takes a parameter x and... let's see later
- What ?
$$\lambda x.(x\ x)$$
takes a parameter x and self-applies it?

Notations

- Instead of $\quad \lambda x_1.(\dots(\lambda x_n.t)\dots) \quad$ we write
$$\lambda x_1 \dots x_n.t$$
- Instead of $\quad (\dots(t\ u_1)\dots u_n) \quad$ we write
$$t\ u_1 \dots u_n$$
or even $\quad t\ \vec{u} \quad$ with $\quad \vec{u} = u_1 \dots u_n$

For instance:

$\lambda c.(\lambda x.c)$	$\lambda cx.c$
$\lambda x.(\lambda y.(\lambda z.((x\ z)\ (y\ z))))$	$\lambda xyz.xz(yz)$

Curryfication and n-ary functions

There is no cartesian product in core λ-calculus.

- A function $\quad (x, y) \mapsto t \quad$ with two parameters is encoded as
$$\lambda x.\lambda y.t \qquad \text{or} \qquad \lambda xy.t$$
- An application $\quad f(x, y) \quad$ of a binary function to two parameters is encoded as
$$f\ x\ y$$

Functions are *curryfied* (tribute to Haskell Curry).
This encoding allows *partial applications.*

Computing with the λ-calculus

Smallest computing block: a function applied to an argument.

$$(\lambda x.t)\ u \quad \longrightarrow \quad t\{x \leftarrow u\}$$

Result :

t where each occurrence of x is replaced by u t$\{x \longleftarrow u\}$

Sample computation

$$
\begin{array}{lll}
& (\lambda xyz.xz\ (yz))\ (\lambda ab.a)\ t\ u & \\
& & \{x \leftarrow \lambda ab.a\} \\
\rightarrow & (\lambda yz.(\lambda ab.a)z\ (yz))\ t\ u & \\
& & \{y \leftarrow t\} \\
\rightarrow & (\lambda z.(\lambda ab.a)z\ (tz))\ u & \\
& & \{z \leftarrow u\} \\
\rightarrow & (\lambda ab.a)u\ (tu) & \\
& & \{a \leftarrow u\} \\
\rightarrow & (\lambda b.u)\ (tu) & \\
& & \{b \leftarrow tu\} \\
\rightarrow & u &
\end{array}
$$

Exercise : reduction

Compute the result of

$$(\lambda xy.yx)\ (\lambda ab.b)\ (\lambda s.stu)$$

Answer

$$
\begin{array}{ll}
& (\lambda xy.yx)\ (\lambda ab.b)\ (\lambda s.stu) \\
\rightarrow & (\lambda y.y\ (\lambda ab.b))\ (\lambda s.stu) \\
\rightarrow & (\lambda s.stu)\ (\lambda ab.b) \\
\rightarrow & (\lambda ab.b)\ t\ u \\
\rightarrow & (\lambda b.b)\ u \\
\rightarrow & u
\end{array}
$$

Exercise : combinatory logic

Combinatory logic (Schönfinkel, 1920 - Curry, 1930) uses the five symbols I, K, S, B, C (called "combinators") and one reduction rule for each.

$$
\begin{array}{rcl}
I\ x & \rightarrow & x \\
K\ x\ y & \rightarrow & x \\
S\ x\ y\ z & \rightarrow & xz\ (yz) \\
B\ x\ y\ z & \rightarrow & x\ (yz) \\
C\ x\ y\ z & \rightarrow & xz\ y
\end{array}
$$

Find λ-terms equivalent to these combinators

Compute the results of the following expressions

1. $S\ K\ K\ x$
2. $S\ (K\ S)\ K$

Answer λ-terms equivalent to combinators

- $I = \lambda x.x$
- $K = \lambda xy.x$
- $S = \lambda xyz.xz(yz)$
- $B = \lambda xyz.x(yz)$
- $C = \lambda xyz.xzy$

Reductions

- $S\ K\ K$ is equivalent to I

$$\begin{array}{rcl} S\ K\ K\ x & \rightarrow & Kx(Kx) \\ & \rightarrow & x \end{array}$$

- $S\ (K\ S)\ K$ is equivalent to B

$$\begin{array}{rcl} S\ (K\ S)\ K\ x\ y\ z & \rightarrow & (K\ S\ x)\ (K\ x)\ y\ z \\ & \rightarrow & S\ (K\ x)\ y\ z \\ & \rightarrow & (K\ x\ z)\ (y\ z) \\ & \rightarrow & x\ (y\ z) \end{array}$$

Dubious replacements / variable capture

How should we resolve the following replacements?

$$(\lambda x.(\lambda x.x))\ y \quad \rightarrow \quad (\lambda x.x)\{x \leftarrow y\}$$

$$(\lambda x.(\lambda y.x))\ y \quad \rightarrow \quad (\lambda y.x)\{x \leftarrow y\}$$

Related: what is the live-range of a variable?

3 Formalization of λ-terms

Set of terms

The set Λ of the λ-terms is *the smallest set* that contains:

1. x for all variable x
2. $\lambda x.t$ if $t \in \Lambda$
3. $t_1\ t_2$ if $t_1 \in \Lambda$ and $t_2 \in \Lambda$

Same definition, stated as an algebraic grammar.

$$t \quad ::= \quad x \quad | \quad \lambda x.t \quad | \quad t_1\ t_2$$

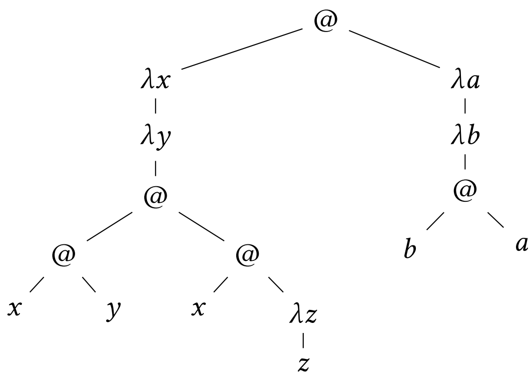
This definition is recursive, and allows *recursive reasoning*.

Term = tree

The expression

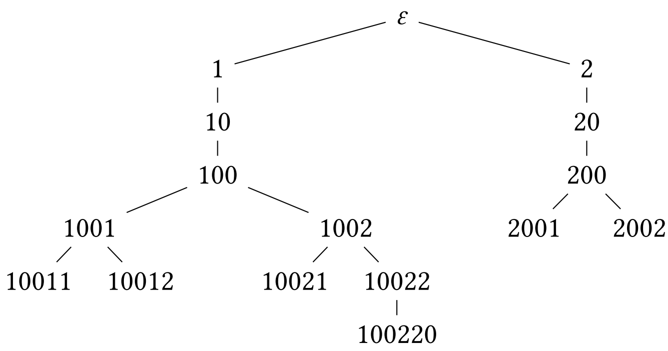
$$(\lambda xy.xy(x(\lambda z.z))\ (\lambda ab.ba)$$

denotes the tree



Positions in a term

Position: word over the alphabet $\{0, 1, 2\}$ denoting a path from the root.



Set $\text{pos}(t)$ of the positions of the term t

$$
\begin{array}{rcl}
\text{pos}(x) & = & \{\varepsilon\} \\
\text{pos}(\lambda x.t) & = & \{\varepsilon\} \quad \cup \quad 0 \cdot \text{pos}(t) \\
\text{pos}(t_1\ t_2) & = & 1 \cdot \text{pos}(t_1) \quad \cup \quad 2 \cdot \text{pos}(t_2)
\end{array}
$$

Encoding in caml

An algebraic datatype for λ-terms

```
type term =
  | Var of string
  | Abs of string * term
  | App of term   * term
```

Encoding of the term $\lambda ab.ba$

```
Abs("a", Abs("b", App(Var "b", Var "a")))
```

Defining functions on lambda-terms

Recursive definition of f, with three cases:

- $f(x)$ — base
- $f(\lambda x.t)$ — using $f(t)$
- $f(t_1\ t_2)$ — using $f(t_1)$ and $f(t_2)$

Examples

$f_@$: number of applications

$$
\begin{array}{rcl}
f_@(x) & = & 0 \\
f_@(\lambda x.t) & = & f_@(t) \\
f_@(t_1\ t_2) & = & 1 + f_@(t_1) + f_@(t_2)
\end{array}
$$

f_v : number of variable occurrences

$$
\begin{array}{rcl}
f_v(x) & = & 1 \\
f_v(\lambda x.t) & = & f_v(t) \\
f_v(t_1\ t_2) & = & f_v(t_1) + f_v(t_2)
\end{array}
$$

Defining a function in caml

Coding $f_@$

```
let rec nb_app = function
  | Var _        -> 0
  | Abs(_, t)    -> nb_app t
  | App(t1, t2) -> 1 + nb_app t1 + nb_app t2
```

Coding f_v

```
let rec nb_var = function
  | Var _        -> 1
  | Abs(_, t)    -> nb_var t
  | App(t1, t2) -> nb_var t1 + nb_var t2
```

Induction principle on lambda-terms

Goal: proving that a property P is true for all λ-terms. Three steps:

- prove $P(x)$ for any variable x
- prove $P(\lambda x.t)$ assuming that $P(t)$ is true
- prove $P(t_1\ t_2)$ assuming that $P(t_1)$ and $P(t_2)$ are both true

Example of inductive reasoning

Goal: for any $t \in \Lambda$, $f_v(t) = 1 + f_@(t)$

- Proof of $P(x)$. By definition, $f_v(x) = 1$ and $f_@(x) = 0$ Then $f_v(x) = 1 + f_@(x)$
- Proof of $P(t) \Longrightarrow P(\lambda x.t)$. Assume $f_v(t) = 1 + f_@(t)$. Then

$$\begin{array}{rcll} f_v(\lambda x.t) & = & f_v(t) & \textit{by definition of } f_v \\ & = & 1 + f_@(t) & \textit{by induction hypothesis} \\ & = & 1 + f_@(\lambda x.t) & \textit{by definition of } f_@ \end{array}$$

- Proof of $P(t_1) \wedge P(t_2) \Longrightarrow P(t_1\ t_2)$. Assume $f_v(t_1) = 1 + f_@(t_1)$ and $f_v(t_2) = 1 + f_@(t_2)$. Then

$$\begin{array}{lll} f_v(t_1\ t_2) & & \\ = & f_v(t_1) + f_v(t_2) & \textit{by definition of } f_v \\ = & 1 + f_@(t_1) + 1 + f_@(t_2) & \textit{by induction hypotheses} \\ = & 1 + (1 + f_@(t_1) + f_@(t_2)) & \\ = & 1 + f_@(t_1\ t_2) & \textit{by definition of } f_@ \end{array}$$

4 Variables and substitutions

A note on variables

The λ-abstraction

$$\lambda x.t$$

introduces a variable x *locally* in t We call it a *bound variable*

In other words:

- the name x is not known outside of t
- seen from the outside, the name x means nothing
- changing the name x does not affect the outside world

Free variables

Variables that can be seen from "outside"

$$\begin{array}{rcl} \mathsf{fv}(x) & = & \{x\} \\ \mathsf{fv}(t_1\ t_2) & = & \mathsf{fv}(t_1) \cup \mathsf{fv}(t_2) \\ \mathsf{fv}(\lambda x.t) & = & \mathsf{fv}(t) \setminus \{x\} \end{array}$$

Term with no free variables: *closed term*, or *combinator*

A name which appears both free and bound in a term:

$$x\ (\lambda x.x)$$

Substitution

Replacing *free* occurrences of x in t by u.

$$t\{x \leftarrow u\}$$

Definition: inductively on the structure of t.

$$\begin{array}{rcl}
y\{x \leftarrow u\} & = & \begin{cases} u & \text{if } x = y \\ y & \text{if } x \neq y \end{cases} \\
(t_1\ t_2)\{x \leftarrow u\} & = & t_1\{x \leftarrow u\}\ t_2\{x \leftarrow u\} \\
(\lambda y.t)\{x \leftarrow u\} & = & \begin{cases} \lambda y.t & \text{if } x = y \\ \lambda y.t\{x \leftarrow u\} & \text{if } x \neq y \text{ and } y \notin \text{fv}(u) \\ \lambda z.t\{y \leftarrow z\}\{x \leftarrow u\} & \text{if } x \neq y \text{ and } y \in \text{fv}(u) \\ & z \text{ new variable} \end{cases}
\end{array}$$

Barendregt's convention

To avoid abuse of names, we consider only terms where

no variable name appears both free and bound in any given subterm

Don't write...	Write... instead
$\lambda x.(x\ (\lambda x.x))$	$\lambda x.(x\ (\lambda y.y))$

Simplified definition for the substitution, relying on the convention

$$\begin{array}{rcl}
y\{x \leftarrow u\} & = & \begin{cases} u & \text{si } x = y \\ y & \text{si } x \neq y \end{cases} \\
(t_1\ t_2)\{x \leftarrow u\} & = & t_1\{x \leftarrow u\}\ t_2\{x \leftarrow u\} \\
(\lambda y.t)\{x \leftarrow u\} & = & \lambda y.t\{x \leftarrow u\}
\end{array}$$

(Un)stability of Barendregt's convention

$$\begin{array}{ll}
(\lambda x.xx)\ (\lambda yz.yz) & \\
\longrightarrow & (\lambda yz.yz)\ (\lambda yz.yz) \\
\longrightarrow & \lambda z.((\lambda yz.zy)z)
\end{array}$$

Preserving Barendregt's convention over reduction requires *changing some variable names during computation*

Bound variables renaming: α-conversion

$$\lambda x.t \quad =_\alpha \quad \lambda y.(t\{x \leftarrow y\}) \qquad\qquad \text{if } y \notin \text{fv}(t)$$

The α-conversion does not change the meaning of a term:

- we can apply it *whenever* we need it

The α-conversion is a *congruence*:

$$\begin{array}{rcl}
t =_\alpha t' & \Longrightarrow & \lambda x.t =_\alpha \lambda x.t' \\
t_1 =_\alpha t_1' & \Longrightarrow & t_1\ t_2 =_\alpha t_1'\ t_2 \\
t_2 =_\alpha t_2' & \Longrightarrow & t_1\ t_2 =_\alpha t_1\ t_2'
\end{array}$$

- we can apply it *wherever* we need it

From now on we assume that any term we work with satisfies Barendregt's convention.

Exercise : bound variables and renaming

Rename some variables of these terms suivants so that they obey Barendregt's convention.

1. $\lambda x.(\lambda x.xy)(\lambda y.xy)$
2. $\lambda xy.x(\lambda y.(\lambda y.y)yz)$

Compute the result of

$$(\lambda f.f\ f)\ (\lambda ab.b\ a\ b)$$

Answer

1. $\lambda x.(\lambda x.xy)(\lambda y.xy) \quad =_\alpha \quad \lambda x.(\lambda z.zy)(\lambda t.xt)$
2. $\lambda xy.x(\lambda y.(\lambda y.y)yz) \quad =_\alpha \quad \lambda xy.x(\lambda a.(\lambda b.b)az)$
3. $$\begin{array}{rcl} (\lambda f.f\ f)\ (\lambda ab.b\ a\ b) & \longrightarrow_\beta & (\lambda ab.b\ a\ b)\ (\lambda ab.b\ a\ b) \\ & \longrightarrow_\beta & \lambda ab.b\ (\lambda ab.b\ a\ b)\ b \\ & =_\alpha & \lambda b.b\ (\lambda xy.y\ x\ y)\ b \end{array}$$

Exercise : free variables and substitution

Prove that

$$\mathsf{fv}(t\{x \leftarrow u\}) \quad \subseteq \quad (\mathsf{fv}(t) \setminus \{x\}) \cup \mathsf{fv}(u)$$

Are these two sets equal?

Answer Proof by induction on the structure of t

- Case where t is a variable
  - case x : $\mathsf{fv}(x\{x \leftarrow u\}) = \mathsf{fv}(u) \subseteq (\mathsf{fv}(t) \setminus \{x\}) \cup \mathsf{fv}(u)$
  - case $y \neq x$: $\mathsf{fv}(y\{x \leftarrow u\}) = \mathsf{fv}(y) = \{y\}$, and $\{y\}$ is indeed a subset of $(\mathsf{fv}(y) \setminus \{x\}) \cup \mathsf{fv}(u) = \{y\} \cup \mathsf{fv}(u)$
- Case where t is an application $t_1\ t_2$. Assume $\mathsf{fv}(t_1\{x \leftarrow u\}) \quad \subseteq \quad (\mathsf{fv}(t_1) \setminus \{x\}) \cup \mathsf{fv}(u)$ and $\mathsf{fv}(t_2\{x \leftarrow u\}) \quad \subseteq \quad (\mathsf{fv}(t_2) \setminus \{x\}) \cup \mathsf{fv}(u)$ (it is our induction hypothesis). Then

$$\begin{array}{lll} \multicolumn{2}{l}{\mathsf{fv}((t_1\ t_2)\{x \leftarrow u\})} & \\ = & \mathsf{fv}((t_1\{x \leftarrow u\})\ (t_2\{x \leftarrow u\})) & \text{by definition of substitution} \\ = & \mathsf{fv}(t_1\{x \leftarrow u\}) \cup \mathsf{fv}(t_2\{x \leftarrow u\}) & \text{by definition of fv} \\ \subseteq & (\mathsf{fv}(t_1) \setminus \{x\}) \cup \mathsf{fv}(u) \cup (\mathsf{fv}(t_2) \setminus \{x\}) \cup \mathsf{fv}(u) & \text{by induction hypothesis} \\ = & (\mathsf{fv}(t_1) \setminus \{x\}) \cup (\mathsf{fv}(t_2) \setminus \{x\}) \cup \mathsf{fv}(u) & \\ = & ((\mathsf{fv}(t_1) \cup \mathsf{fv}(t_2)) \setminus \{x\}) \cup \mathsf{fv}(u) & \\ = & (\mathsf{fv}(t_1\ t_2) \setminus \{x\}) \cup \mathsf{fv}(u) & \end{array}$$

- Case where t is a λ-abstraction $\lambda y.t_0$. Assume $x \neq y$ and $y \notin \mathsf{fv}(u)$ (if not, α-rename it). Assume $\mathsf{fv}(t_0\{x \leftarrow u\}) \quad \subseteq \quad (\mathsf{fv}(t_0) \setminus \{x\}) \cup \mathsf{fv}(u)$ (induction hypothesis). Then

$$\begin{array}{lll} \multicolumn{2}{l}{\mathsf{fv}((\lambda y.t_0)\{x \leftarrow u\})} & \\ = & \mathsf{fv}(\lambda y.(t_0\{x \leftarrow u\})) & \text{since } x \neq y \text{ and } y \notin \mathsf{fv}(u) \\ = & \mathsf{fv}(t_0\{x \leftarrow u\}) \setminus \{y\} & \\ \subseteq & ((\mathsf{fv}(t_0) \setminus \{x\}) \cup \mathsf{fv}(u)) \setminus y & \text{induction hypothesis} \\ = & ((\mathsf{fv}(t_0) \setminus \{x\} \setminus \{y\}) \cup (\mathsf{fv}(u) \setminus y) & \\ = & ((\mathsf{fv}(t_0) \setminus \{x\} \setminus \{y\}) \cup \mathsf{fv}(u) & \text{since } y \notin \mathsf{fv}(u) \\ = & ((\mathsf{fv}(t_0) \setminus \{y\} \setminus \{x\}) \cup \mathsf{fv}(u) & \\ = & (\mathsf{fv}(\lambda y.t_0) \setminus x) \cup \mathsf{fv}(u) & \end{array}$$

The sets are not equal: if $x \notin \mathsf{fv}(t)$ then u disappears in $t\{x \leftarrow u\}$, together with its free variables.

5 Formalisation of the reduction

β-reduction

Application of a function to an argument

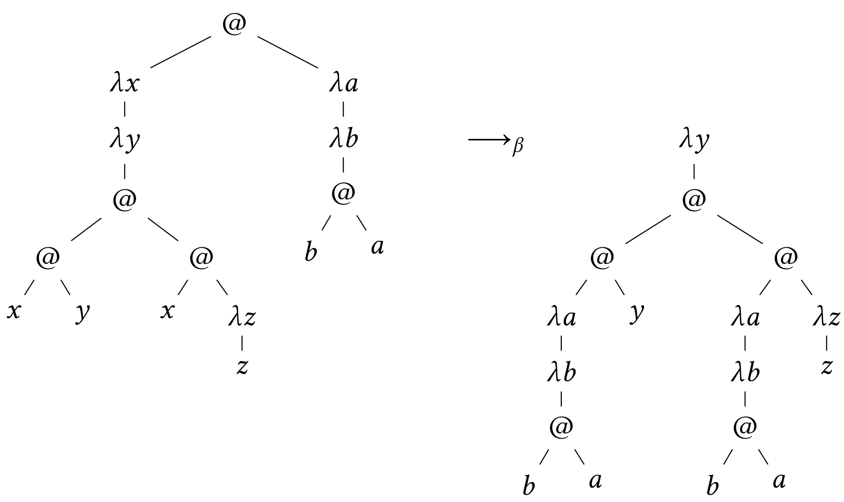
$$(\lambda x.t)\ u$$

The result if given by the function body, in which the formal parameter x is linked to the argument u.

$$(\lambda x.t)\ u \quad \longrightarrow_\beta \quad t\{x \leftarrow u\}$$

where $t\{x \leftarrow u\}$ denotes substitution *without capture*

β-reduction, pictured on trees



β-reduction, programmed in caml

Function for reducing a β-redex

```
let beta_reduction = function
  | App(Abs(x, t), u) -> subst t x u
  | _ -> failwith "not␣a␣beta-redex"
```

Auxiliary function : `subst` $t\ x\ u$ computes $t\{x \leftarrow u\}$

```
let rec subst t x u = match t with
  | Var y         -> if x = y then u else t
  | App(t1, t2) -> App(subst t1 x u,
                         subst t2 x u)
  | Abs(y, t)    -> (* renaming ? *)
```

Congruence

The β-reduction rule can be applied anywhere in a term. This can be formalized using inference rules.

$$\frac{}{(\lambda x.t)\ u \quad \longrightarrow_\beta \quad t\{x \leftarrow u\}}$$

$$\frac{t \quad \longrightarrow_\beta \quad t'}{t\ u \quad \longrightarrow_\beta \quad t'\ u} \qquad\qquad \frac{u \quad \longrightarrow_\beta \quad u'}{t\ u \quad \longrightarrow_\beta \quad t\ u'}$$

$$\frac{t \quad \longrightarrow_\beta \quad t'}{\lambda x.t \quad \longrightarrow_\beta \quad \lambda x.t'}$$

Position of a reduction

Write

$$t \xrightarrow{p}_\beta t'$$

when t reduces to t' by contracting a redex at position p

$$\frac{}{(\lambda x.t)\ u \xrightarrow{\varepsilon}_\beta t\{x \leftarrow u\}}$$

$$\frac{t \xrightarrow{p}_\beta t'}{t\ u \xrightarrow{1\cdot p}_\beta t'\ u} \qquad \frac{u \xrightarrow{p}_\beta u'}{t\ u \xrightarrow{2\cdot p}_\beta t\ u'}$$

$$\frac{t \xrightarrow{p}_\beta t'}{\lambda x.t \xrightarrow{0\cdot p}_\beta \lambda x.t'}$$

Justifying a reduction using a derivation tree

$$\cfrac{\cfrac{\cfrac{\cfrac{}{(\lambda y.zy)\ x \xrightarrow{\varepsilon} zx}}{x\ ((\lambda y.zy)\ x) \xrightarrow{2} x\ (zx)}}{\lambda x.(x\ ((\lambda y.zy)\ x)) \xrightarrow{02} \lambda x.(x\ (zx))}}{(\lambda x.x\ ((\lambda y.zy)x))\ z \xrightarrow{102} (\lambda x.x\ (zx))\ z}$$

Inductive reasoning on a reduction

Since the reduction relation $t \longrightarrow_\beta t'$ is defined by inference rules, there is an associated inductive reasoning principle. On can prove that a property P is such that

$$\forall t, t', \quad t \longrightarrow_\beta t' \quad \Longrightarrow \quad P(t, t')$$

by simply checking the following four points:

- $P((\lambda x.t)u,\ t\{x \leftarrow u\})$ for any x, t and u *base case*
- $P(tu,\ t'u)$ for any t, t' and u such that $P(t, t')$ *inductive case*
- $P(tu,\ tu')$ for any t, u and u' such that $P(u, u')$ *another inductive case*
- $P(\lambda x.t,\ \lambda x.t')$ for any x, t and t' such that $P(t, t')$ *yet another inductive case*

Notice that these four conditions are quite similar to the four inference rules

Inductive reasoning on reduction

Reduction does not generate free variables.

$$\text{If} \quad t \longrightarrow t' \quad \text{, then} \quad \mathsf{fv}(t') \subseteq \mathsf{fv}(t)$$

Proof by induction on the derivation of $t \longrightarrow t'$.

- Case $(\lambda x.t)\ u \longrightarrow t\{x \leftarrow u\}$. We already proved: $\mathsf{fv}(t\{x \leftarrow u\}) \subseteq (\mathsf{fv}(t) \setminus \{x\}) \cup \mathsf{fv}(u)$. Moreover, we have

$$\begin{array}{rcl} \mathsf{fv}((\lambda x.t)\ u) & = & \mathsf{fv}(\lambda x.t) \cup \mathsf{fv}(u) \\ & = & (\mathsf{fv}(t) \setminus \{x\}) \cup \mathsf{fv}(u) \end{array}$$

- Case $t\ u \longrightarrow t'\ u$ with $t \longrightarrow t'$. Then

$$\begin{array}{rcll} \mathsf{fv}(t'\ u) & = & \mathsf{fv}(t') \cup \mathsf{fv}(u) & \text{by definition} \\ & \subseteq & \mathsf{fv}(t) \cup \mathsf{fv}(u) & \text{by induction hypothesis} \\ & = & \mathsf{fv}(t\ u) & \text{by definition} \end{array}$$

- Case $t\ u' \longrightarrow t\ u'$ with $u \longrightarrow u'$ similar.
- Case $\lambda x.t \longrightarrow \lambda x.t'$ with $t \longrightarrow t'$. Then

$$\begin{array}{rcll} \mathsf{fv}(\lambda x.t') & = & \mathsf{fv}(t') \setminus \{x\} & \text{by definition} \\ & \subseteq & \mathsf{fv}(t) \setminus \{x\} & \text{by induction hypothesis} \\ & = & \mathsf{fv}(\lambda x.t) & \text{by definition} \end{array}$$

Reduction sequences

$\longrightarrow_\beta$ one step

$\longrightarrow_\beta^*$ reflexive transitive closure: 0, 1 or many steps

$\longleftrightarrow_\beta$ symmetric closure: one step, forward or backward

$=_\beta$ reflexive, symmetric, transitive closure (equivalence)

Additional (optional) rule : η

Depending on what we want to model, can be used in both directions:

- η-contraction

$$\lambda x.(t\ x) \quad \longrightarrow_\eta \quad t$$

- η-expansion

$$t \quad \longrightarrow_\eta \quad \lambda x.(t\ x)$$

Related to *extensional equality* (Leibniz equality)

Alternative formalization: reduction in contexts

Focus on the redex r reduced in a term t

$$t = C[r] \qquad \longrightarrow \qquad C[r'] = t'$$

with $r = (\lambda x.u)v$ and $r' = u\{x \longleftarrow v\}$

C is a *context*: a term with *one* hole

$$C \quad ::= \quad \square \quad | \quad C\ t \quad | \quad t\ C \quad | \quad \lambda x.C$$

$C[u]$ is the result of filling the hole of C with the term u

Exercise: contexts and subterms

Here are some decompositions of $\lambda x.(x\ \lambda y.xy)$ into a context and a term $C[u]$

C	$\square$	$\lambda x.\square$	$\lambda x.(\square\ \lambda y.xy)$	$\lambda x.(x\ \square)$	...
u	$\lambda x.(x\ \lambda y.xy)$	$x\ \lambda y.xy$	x	$\lambda y.xy$	...

What are the other possible decompositions?

We already showed that

$$(\lambda x.x\ ((\lambda y.zy)x))\ z \quad \longrightarrow \quad (\lambda x.x\ (zx))\ z$$

What are the context and the redex associated to this reduction?

Answer Other decompositions of $\lambda x.(x\ \lambda y.xy)$

C	$\lambda x.(x\ (\lambda y.\square))$	$\lambda x.(x\ (\lambda y.\square\ y))$	$\lambda x.(x\ (\lambda y.x\ \square))$
u	xy	x	y

Decomposition of the reduction:

$$C[(\lambda y.zy)x] \quad \longrightarrow \quad C[zx]$$

with $C = (\lambda x.x\ \square)\ z$

Exercise: equivalence of the two formalizations (first way)

Prove that if

$$t \longrightarrow_\beta t'$$

then there are C, x, u, v such that

$$t = C[(\lambda x.u)v] \qquad \text{et} \qquad t' = C[u\{x \leftarrow v\}]$$

Answer Proof by induction on the derivation of $t \longrightarrow_\beta t'$.

- Base case $t = (\lambda x.u)v \longrightarrow_\beta u\{x \leftarrow v\} = t'$. Straightforward conclusion with the context $\square$
- Case $t = t_1 t_2 \longrightarrow_\beta t_1' t_2 = t'$ with $t_1 \longrightarrow_\beta t_1'$. Assume there are C_1, x, u and v such that $t_1 = C_1[(\lambda x.u)v]$ and $t_1' = C_1[u\{x \leftarrow v\}]$ (induction hypothesis). Then conclude with $C = C_1\ t_2$
- Case $t = t_1 t_2 \longrightarrow_\beta t_1 t_2' = t'$ with $t_2 \longrightarrow_\beta t_2'$ similar, using context $C = t_1\ C_2$
- Case $t = \lambda y.t_0 \longrightarrow_\beta \lambda y.t_0' = t'$ with $t_0 \longrightarrow_\beta t_0'$ similar, using context $C = \lambda y.C_0$

Pure λ-calculus: summary

Minimalistic formalism

- Variables
- λ-abstraction
- Application
- α-renaming
- β-reduction

Theoretically, we do not need anything else! *see chapter on λ-computability*

6 Extended λ-calculi

PCF: *Programming with Computable Functions*

The λ-calculus can be extended with various programming features we want to study. Pick your favorite:

- integer arithmetic
- booleans and conditionals
- data structures
- recursive functions
- ...

PCF is a standard package of such extensions

Extending the λ-calculus

Ingredients

- new syntax
- reduction rules
- extended definitions (e.g. substitution)
- extended proofs

Integer arithmetic

New shapes of terms

$$
\begin{array}{lcll}
t & ::= & \dots & \\
 & | & n & \text{integer} \\
 & | & t_1 \ op \ t_2 & \text{binary operation } \oplus, \ominus, \dots
\end{array}
$$

New base reduction rules

$$n_1 \oplus n_2 \quad \longrightarrow \quad n \qquad \text{with} \quad n = n_1 + n_2$$

New congruence rules

$$\frac{t_1 \longrightarrow t_1'}{t_1 \oplus t_2 \longrightarrow t_1' \oplus t_2} \qquad \frac{t_2 \longrightarrow t_2'}{t_1 \oplus t_2 \longrightarrow t_1 \oplus t_2'}$$

Extended definitions

$$
\begin{array}{rcl}
\mathsf{fv}(t_1 \ op \ t_2) & = & \mathsf{fv}(t_1) \cup \mathsf{fv}(t_2) \\
(t_1 \ op \ t_2)\{x \leftarrow u\} & = & (t_1\{x \leftarrow u\}) \ op \ (t_2\{x \leftarrow u\})
\end{array}
$$

Booleans and conditionals

New shapes of terms

$$
\begin{array}{lcll}
t & ::= & \dots & \\
 & | & \mathsf{T} & \text{true} \\
 & | & \mathsf{F} & \text{false} \\
 & | & \mathsf{isZero}(t) & \text{test} \\
 & | & \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 & \text{conditional expression}
\end{array}
$$

New base rules

$$
\begin{array}{rcll}
\mathsf{isZero}(0) & \longrightarrow & \mathsf{T} & \\
\mathsf{isZero}(n) & \longrightarrow & \mathsf{F} & n \neq 0 \\
\mathsf{if}\ \mathsf{T}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 & \longrightarrow & t_1 & \\
\mathsf{if}\ \mathsf{F}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 & \longrightarrow & t_2 &
\end{array}
$$

+ new congruence rules

Pairs

New shapes of terms

$$
\begin{array}{lcll}
t & ::= & \dots & \\
 & | & \langle t_1, t_2 \rangle & \text{pair} \\
 & | & \pi_1(t) & \text{left projection} \\
 & | & \pi_2(t) & \text{right projection}
\end{array}
$$

New base rules

$$
\begin{array}{rcl}
\pi_1(\langle t_1, t_2 \rangle) & \longrightarrow & t_1 \\
\pi_2(\langle t_1, t_2 \rangle) & \longrightarrow & t_2
\end{array}
$$

+ new congruence rules

Linked lists

New shapes of terms

$$
\begin{array}{rcll}
t & ::= & \ldots & \\
& | & \mathsf{Nil} & \text{empty list} \\
& | & t_1 :: t_2 & \text{combine an element (head) and a list (tail)} \\
& | & \mathsf{isNil}(t) & \text{test} \\
& | & \mathsf{hd}(t) & \text{head element} \\
& | & \mathsf{tl}(t) & \text{tail of the list}
\end{array}
$$

New base rules

$$
\begin{array}{rcl}
\mathsf{isNil}(\mathsf{Nil}) & \longrightarrow & \mathsf{T} \\
\mathsf{isNil}(t_1 :: t_2) & \longrightarrow & \mathsf{F} \\
\mathsf{hd}(t_1 :: t_2) & \longrightarrow & t_1 \\
\mathsf{tl}(t_1 :: t_2) & \longrightarrow & t_2
\end{array}
$$

+ congruence rules

Recursion

New shapes of terms

$$
\begin{array}{rcll}
t & ::= & \ldots & \\
& | & \mathsf{Fix}(t) & \text{fixed point}
\end{array}
$$

New base rules

$$\mathsf{Fix}(t) \longrightarrow t\ (\mathsf{Fix}(t))$$

+ congruence rules

Exercise : extended reduction

Compute the value of the expression

$$\mathsf{Fix}(\ \lambda f s.\mathsf{if}\ \mathsf{isNil}(s)\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \oplus (f(\mathsf{tl}(s)))\ )\ (2::4::8::\mathsf{Nil})$$

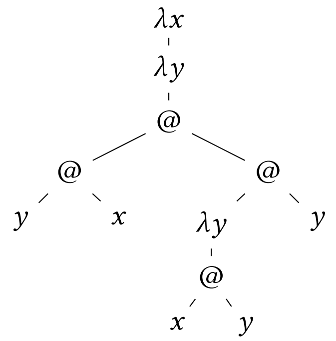
Answer. Write $F = \lambda f s.\mathsf{if}\ \mathsf{isNil}(s)\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \oplus (f(\mathsf{tl}(s)))$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{Fix}(F)\ (2::4::8::\mathsf{Nil})} \\
\longrightarrow & F\ (\mathsf{Fix}(F))\ (2::4::8::\mathsf{Nil}) \\
\longrightarrow & (\lambda s.\mathsf{if}\ \mathsf{isNil}(s)\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \oplus (\mathsf{Fix}(F))(\mathsf{tl}(s)))\ (2::4::8::\mathsf{Nil}) \\
\longrightarrow & \mathsf{if}\ \mathsf{isNil}(2::4::8::\mathsf{Nil})\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \oplus (\mathsf{Fix}(F))(\mathsf{tl}(2::4::8::\mathsf{Nil})) \\
\longrightarrow & \mathsf{if}\ \mathsf{F}\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \oplus (\mathsf{Fix}(F))(\mathsf{tl}(2::4::8::\mathsf{Nil})) \\
\longrightarrow & 1 \oplus (\mathsf{Fix}(F))(\mathsf{tl}(2::4::8::\mathsf{Nil})) \\
\longrightarrow & 1 \oplus (\mathsf{Fix}(F))(4::8::\mathsf{Nil}) \\
\ldots & \\
\longrightarrow & 1 \oplus 1 \oplus 1 \oplus (\mathsf{Fix}(F)\ \mathsf{Nil}) \\
\longrightarrow & 1 \oplus 1 \oplus 1 \oplus (F\ (\mathsf{Fix}(F))\ \mathsf{Nil}) \\
\longrightarrow & 1 \oplus 1 \oplus 1 \oplus ((\lambda s.\mathsf{if}\ \mathsf{isNil}(s)\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \oplus (\mathsf{Fix}(F))(\mathsf{tl}(s)))\ \mathsf{Nil}) \\
\longrightarrow & 1 \oplus 1 \oplus 1 \oplus (\mathsf{if}\ \mathsf{isNil}(\mathsf{Nil})\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \oplus (\mathsf{Fix}(F))(\mathsf{tl}(\mathsf{Nil}))) \\
\longrightarrow & 1 \oplus 1 \oplus 1 \oplus 0 \\
\longrightarrow & 1 \oplus 1 \oplus 1 \\
\longrightarrow & 1 \oplus 2 \\
\longrightarrow & 3
\end{array}
$$

7 de Bruijn notation

Use numbers instead of variable names

$\lambda x.\lambda y.(y\ x\ ((\lambda y.xy)\ y))$



Replace each variable occurrence with the number of λ between the occurrence and its binder

$$\lambda.\lambda.0\ 1\ ((\lambda.20)\ 0)$$

What we gain: the need for variable renamings disappears

de Bruijn, in caml

λ-terms with de Bruijn indices

```
type term =
  | Var of int
  | App of term * term
  | Abs of term
```

Encoding of the term $\lambda.\lambda.0\ 1\ ((\lambda.20)\ 0)$

```
Abs(Abs(App(App(Var 0, Var 1),
            App(Abs(App(Var 2, Var 0)),
                Var 0))))
```

Substitutions and indices

β-reduction

- substitution of 0 (occurrences bound by the λ in the redex)

$$(\lambda.0\ (\lambda.0\ 1))\ t \quad \longrightarrow_\beta \quad t\ (\lambda.0\ t)$$

- other indices under the λ-abstraction of the redex should be adjusted (-1)

$$(\lambda.0\ 1\ (\lambda.0\ 1))\ t \quad \not\longrightarrow_\beta \quad t\ 1\ (\lambda.0\ t)$$

  il faut les décrementer

- indices in the substituted argument should also be adjusted each time we cross a λ (+1)

$$(\lambda.0\ 1\ (\lambda.0\ 1))\ 0 \quad \not\longrightarrow_\beta \quad 0\ 1\ (\lambda.0\ 0)$$

Substitution, in caml

Substitution of the index i

```
let rec subst t i u = match t with
  | Var j -> if i=j then u
         else if i<j then Var (j-1)
         else t
  | App(t1,t2) -> App(subst t1 i u,
                      subst t2 i u)
  | Abs t -> let u' = shift 0 u in
             Abs (subst t (i+1) u')
```

Auxiliary function: shift indices greater of equal to k

```
let rec shift k u = match u with
  | Var j -> if   k<=j
             then Var (j+1)
             else u
  | App(t1, t2) -> App(shift k t1,
                       shift k t2)
  | Abs t -> Abs (shift (k+1) t)
```

Exercise: de Bruijn notation

Write the following terms using de Bruijn indices

1. $\lambda x.(\lambda x.xy)(\lambda y.xy)$
2. $\lambda xy.x(\lambda y.(\lambda y.y)yz)$

Write the following term using de Bruijn indices, then reduce it

$$(\lambda f.f\ f)\ (\lambda ab.b\ a\ b)$$

Answer

1. $\lambda.(\lambda.02)(\lambda.10)$
2. $\lambda.\lambda.1(\lambda.(\lambda.0)03$
3. $$\begin{aligned}(\lambda.00)\ (\lambda.\lambda.010) &\rightarrow (\lambda.\lambda.010)\ (\lambda.\lambda.010)\\ &\rightarrow \lambda.0(\lambda.\lambda.010)0\end{aligned}$$

Homework – *write it down and send it to me before next course*

Prove that if $x \neq y$ *and* $x \notin fv(v)$ *then*

$$t\{x \leftarrow u\}\{y \leftarrow v\} \quad = \quad t\{y \leftarrow v\}\{x \leftarrow u\{y \leftarrow v\}\}$$