Lambda-calculus
and programming language semantics

Christine Paulin
Course material by Thibaut Balabonski @ UPSay

ecampus site :

`https://ecampus.paris-saclay.fr/course/view.php?id=199492`

Outline

Lambda-calculus and (functional) programming language semantics

1. Pure lambda calculus
   - syntax, alpha, beta and eta-conversion
2. Reduction strategies
   - confluence
3. λ-computability
   - Representation of data and recursive functions
4. Simply-typed λ-calculus
   - type safety, normalisation, Curry-Howard correspondence
5. Implementing λ-calculus
6. Extra-course (to be confirmed) : second-order λ-calculus

Evaluation

- Homework (25%)
- Final exam (75%)

References

Hendrik Pieter Barendregt.
The lambda calculus - its syntax and semantics, volume 103 of *Studies in logic and the foundations of mathematics*.
North-Holland, 1985.

J. Roger Hindley and Jonathan P. Seldin.
Introduction to Combinators and Lambda-Calculus.
Cambridge University Press, 1986.

Jean-Louis Krivine.
Lambda-calculus, types and models.
Ellis Horwood series in computers and their applications.
Masson, 1993.
https://cel.hal.science/cel-00574575.

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* (undecidability of first-order logic).

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

Terms (expressions)

Formed with only 3 constructions :

- 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

$$\lambda x.x$$

- Constant functions generator

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

- Distribution

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

- What ?

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

Notations

- Instead of $\lambda x_1.(\ldots(\lambda x_n.t)\ldots)$ we write

$$\lambda x_1 \ldots x_n.t$$

- Instead of $(\ldots(t\ u_1)\ldots u_n)$ we write

$$t\ u_1 \ldots u_n$$

or even $t\ \vec{u}$ with $\vec{u} = u_1 \ldots u_n$

For instance:

$\lambda c.(\lambda x.c)$	$\lambda c\, x.c$
$\lambda x.(\lambda y.(\lambda z.((x\ z)\ (y\ z))))$	$\lambda x\, y\, z.x\ z\ (y\ z)$

Curryfication and n-ary functions

There is no cartesian product in core λ-calculus.

- A function $(x, y) \mapsto t$ with two parameters is encoded as

$$\lambda x.\lambda y.t \qquad \text{or} \qquad \lambda x\, y.t$$

- An application $f(x, y)$ 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.

$$(\lambda x.t)\ u \quad \rightarrow \quad ???$$

Result :

t where each occurrence of x is replaced by u

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

Sample computation

$$
\begin{aligned}
& (\lambda x\, y\, z.x\, z\, (y\, z))\ (\lambda a\, b.a)\ t\ u && \\
\rightarrow \quad & (\lambda y\, z.(\lambda a\, b.a) z\ (y\, z))\ t\ u && \{x \leftarrow \lambda a\, b.a\} \\
\rightarrow \quad & (\lambda z.(\lambda a\, b.a)\ z\ (t\, z))\ u && \{y \leftarrow t\} \\
\rightarrow \quad & (\lambda a\, b.a) u\ (t\ u) && \{z \leftarrow u\} \\
\rightarrow \quad & (\lambda b.u)\ (t\ u) && \{a \leftarrow u\} \\
\rightarrow \quad & u && \{b \leftarrow t\ u\}
\end{aligned}
$$

Exercise : reduction

Compute the result of

$$(\lambda x\, y.y\, x)\ (\lambda a\, b.b)\ (\lambda s.s\, t\, u)$$

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.

$$\begin{aligned} I\ x &\to x \\ K\ x\ y &\to x \\ S\ x\ y\ z &\to x\ z\ (y\ z) \\ B\ x\ y\ z &\to x\ (y\ z) \\ C\ x\ y\ z &\to x\ z\ y \end{aligned}$$

Find λ-terms equivalent to these combinators

Compute the results of the following expressions

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

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?

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.

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

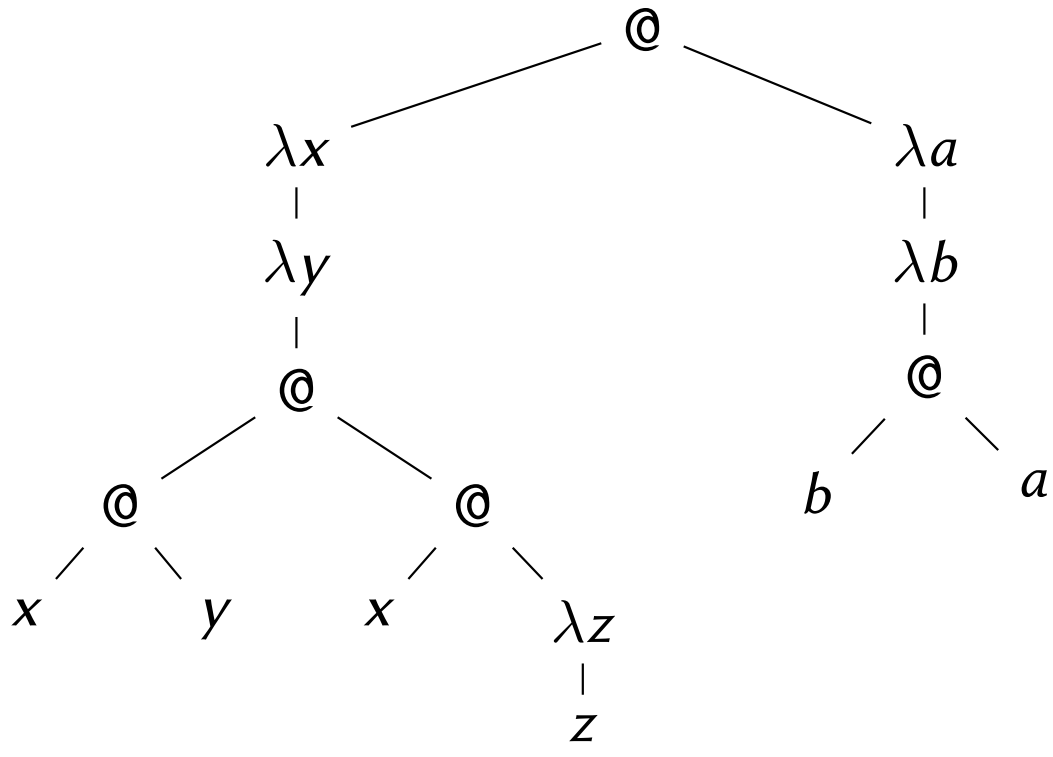
This definition is recursive, and allows *recursive reasoning*.

Term = tree

The expression

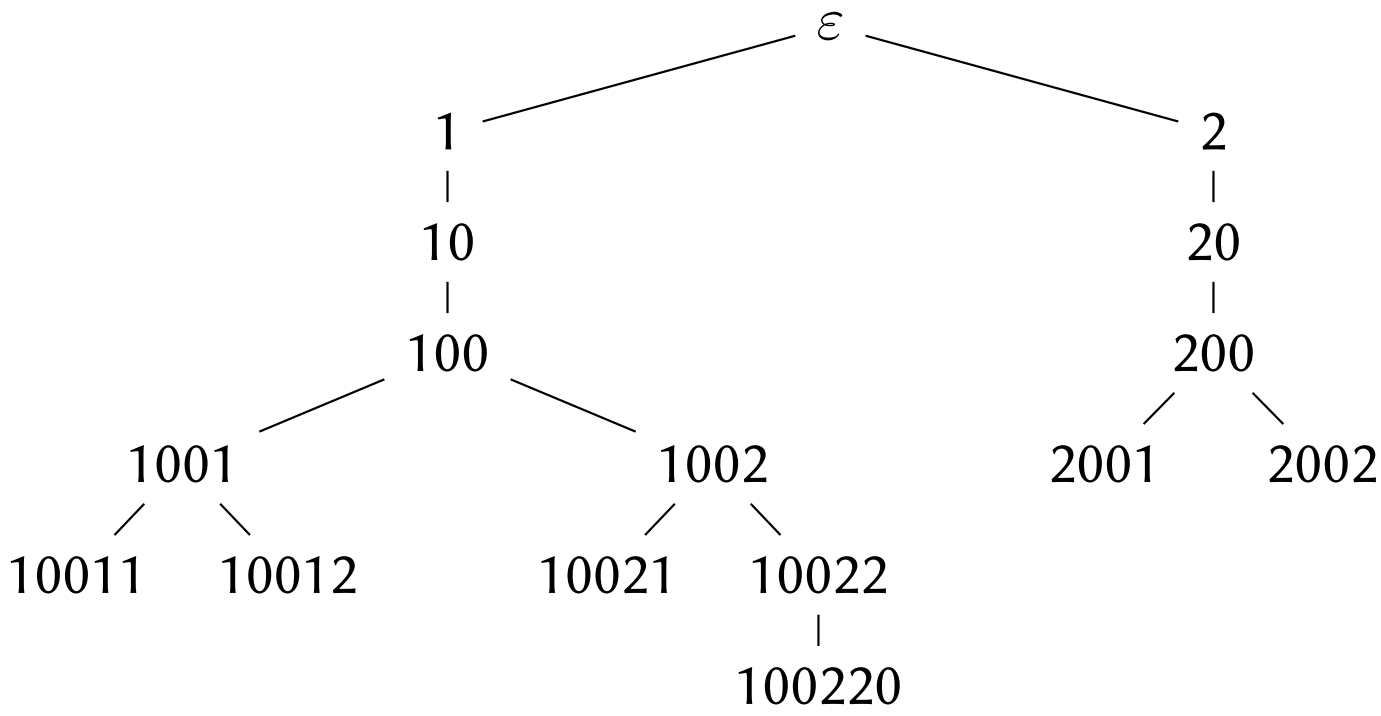
$$\lambda x\, y.x\, y(x(\lambda z.z))\ (\lambda a\, b.b\, a)$$

denotes the tree



Positions in a term

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



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

$$\begin{aligned} \mathrm{pos}(x) &= \{\varepsilon\} \\ \mathrm{pos}(\lambda x.t) &= \{\varepsilon\} \cup 0 \cdot \mathrm{pos}(t) \\ \mathrm{pos}(t_1\ t_2) &= 1 \cdot \mathrm{pos}(t_1) \cup 2 \cdot \mathrm{pos}(t_2) \end{aligned}$$

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 $\lambda a\, b.b\, a$

```
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

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

f_v : number of variable occurrences

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

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 $t \in \Lambda$, $f_v(t) = 1 + f_@(t)$

$P(t)$ is $f_v(t) = 1 + f_@(t)$

- Proof of $P(x)$.
  By definition, $f_v(x) = 1$ and $f_@(x) = 0$
  Then $f_v(x) = 1 + f_@(x)$
- Proof of $P(t) \Rightarrow P(\lambda x.t)$.
  Assume $f_v(t) = 1 + f_@(t)$.
  Then
  $$\begin{array}{lcll} f_v(\lambda x.t) & = & f_v(t) & \textit{by definition of } f_v \\ & = & 1 + f_@(t) & \textit{by induction hypothesis} \\ & = & 1 + f_@(\lambda x.t) & \textit{by definition of } f_@ \end{array}$$
- Proof of $P(t_1) \wedge P(t_2) \Rightarrow P(t_1\ t_2)$.
  Assume $f_v(t_1) = 1 + f_@(t_1)$ and $f_v(t_2) = 1 + f_@(t_2)$.
  Then
  $$\begin{array}{lcll} f_v(t_1\ t_2) & & & \\ = & f_v(t_1) + f_v(t_2) & \textit{by definition of } f_v \\ = & 1 + f_@(t_1) + 1 + f_@(t_2) & \textit{by induction hypotheses} \\ = & 1 + (1 + f_@(t_1) + f_@(t_2)) & \\ = & 1 + f_@(t_1\ t_2) & \textit{by definition of } f_@ \end{array}$$

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"

$$\begin{aligned} \mathrm{fv}(x) &= \{x\} \\ \mathrm{fv}(t_1\ t_2) &= \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2) \\ \mathrm{fv}(\lambda x.t) &= \mathrm{fv}(t) \setminus \{x\} \end{aligned}$$

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.

$$y\{x \leftarrow u\} = \begin{cases} u & \text{if } x = y \\ y & \text{if } x \neq y \end{cases}$$

$$(t_1\ t_2)\{x \leftarrow u\} = t_1\{x \leftarrow u\}\ t_2\{x \leftarrow u\}$$

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

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 $t\{x \leftarrow u\}$, relying on the convention (for t and u together)

$$y\{x \leftarrow u\} = \begin{cases} u & \text{si } x = y \\ y & \text{si } x \neq y \end{cases}$$

$$(t_1\ t_2)\{x \leftarrow u\} = t_1\{x \leftarrow u\}\ t_2\{x \leftarrow u\}$$

$$(\lambda y.t)\{x \leftarrow u\} = \lambda y.t\{x \leftarrow u\}$$

(Un)stability of Barendregt's convention

$$
\begin{aligned}
& (\lambda x.x\,x)\ (\lambda y\,z.y\,z) \\
\rightarrow\quad & (\lambda y\,z.y\,z)\ (\lambda y\,z.y\,z) \\
\rightarrow\quad & \lambda z.((\lambda y\,z.z\,y)z)
\end{aligned}
$$

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

Bound variables renaming: α-conversion

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

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 so that they obey Barendregt's convention.

1. $\lambda x.(\lambda x.x\ y)(\lambda y.x\ y)$
2. $\lambda x\ y.x(\lambda y.(\lambda y.y)y\ z)$

Compute the result of

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

Exercise : free variables and substitution

Prove that

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

Are these two sets equal?

β-reduction

Application of a function to an argument

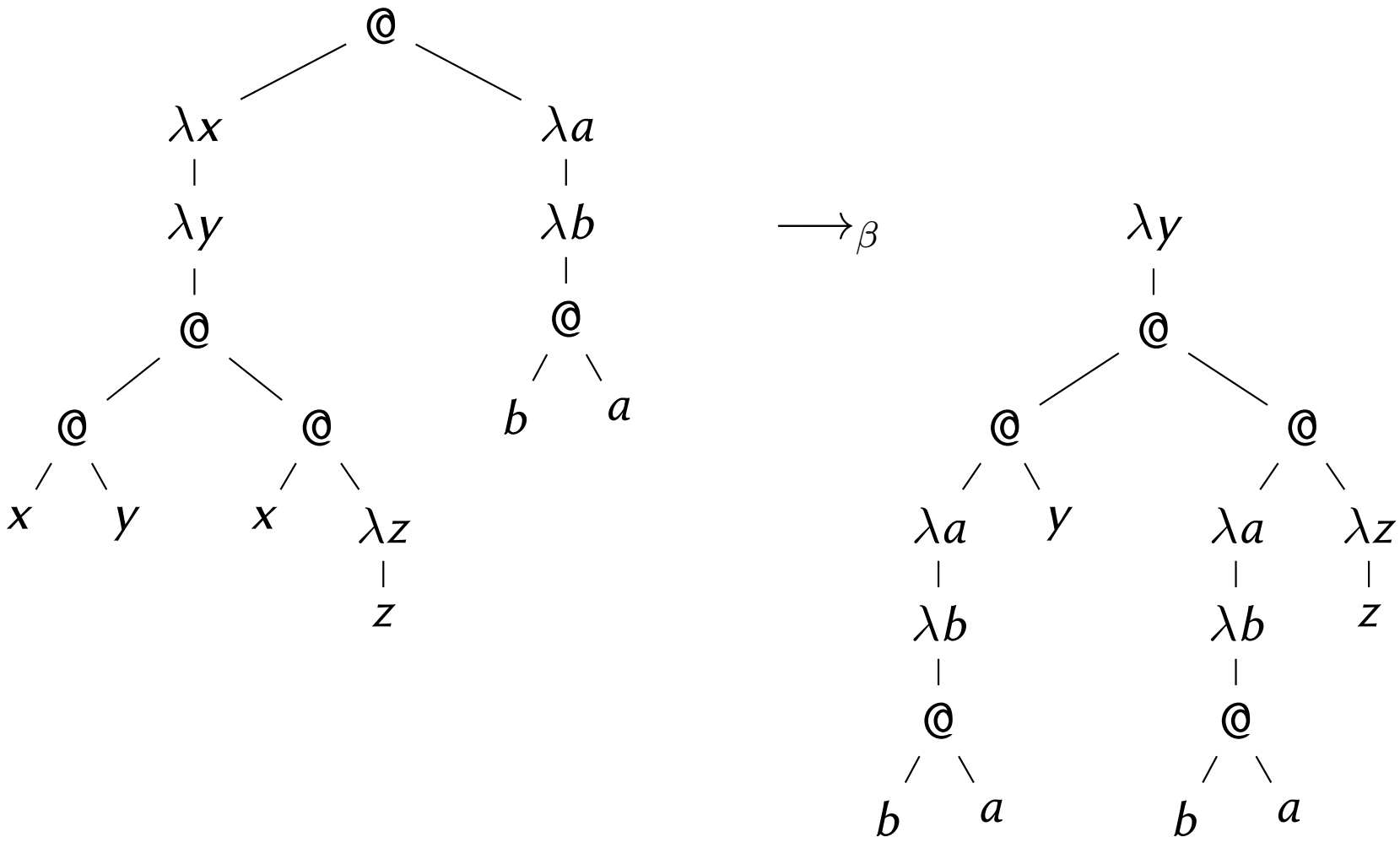
$$(\lambda x.t)\ u$$

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

$$(\lambda x.t)\ u \qquad \rightarrow_\beta \qquad t\{x \leftarrow u\}$$

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.

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

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

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

Position of a reduction

Write

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

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

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

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

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

Justifying a reduction using a derivation tree

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

Inductive reasoning on a reduction

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

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

by simply checking the following four points:

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

$$\text{If} \quad t \to t', \text{ then} \quad \mathrm{fv}(t') \subseteq \mathrm{fv}(t)$$

Proof by induction on the derivation of $t \to t'$,
$P(t, t')$ is $\mathrm{fv}(t') \subseteq \mathrm{fv}(t)$

- Case $(\lambda x.t)\ u \to t\{x \leftarrow u\}$.
  We already proved: $\mathrm{fv}(t\{x \leftarrow u\}) \subseteq (\mathrm{fv}(t) \setminus \{x\}) \cup \mathrm{fv}(u)$.
  Moreover, we have $\begin{aligned} \mathrm{fv}((\lambda x.t)\ u) &= \mathrm{fv}(\lambda x.t) \cup \mathrm{fv}(u) \\ &= (\mathrm{fv}(t) \setminus \{x\}) \cup \mathrm{fv}(u) \end{aligned}$
- Case $t\ u \to t'\ u$ with $t \to t'$. Then

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

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

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

Reduction sequences

$\rightarrow_\beta$ one step

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

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

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

Additional (optional) rule : η

Link between a function f and the function $\lambda x.(f\ x)$?

- if f is $\lambda y.t$ it is the same by β-conversion
- otherwise ?

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

- η-contraction

$$\lambda x.(f\ x) \quad \to_\eta \quad f$$

- η-expansion

$$f \quad \to_\eta \quad \lambda x.(f\ x)$$

In both cases, there is a side condition : $x \notin \mathrm{fv}(f)$

Alternative formalization: reduction in contexts

Focus on the redex r reduced in a term t

$$t = \mathcal{C}[r] \qquad \rightarrow \qquad \mathcal{C}[r'] = t'$$

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

$\mathcal{C}$ is a *context*: a term with *one* hole

$$\mathcal{C} \quad ::= \quad \Box \quad | \quad \mathcal{C}\ t \quad | \quad t\ \mathcal{C} \quad | \quad \lambda x.\mathcal{C}$$

$\mathcal{C}[u]$ is the result of filling the hole of $\mathcal{C}$ with the term u

Exercise: contexts and subterms

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

$\mathcal{C}$	$\Box$	$\lambda x.\Box$	$\lambda x.(\Box\ \lambda y.x\ y)$	$\lambda x.(x\ \Box)$	...
u	$\lambda x.(x\ \lambda y.x\ y)$	$x\ \lambda y.x\ y$	x	$\lambda y.x\ y$	...

What are the other possible decompositions?

We already showed that

$$(\lambda x.x\ ((\lambda y.z\ y)x))\ z \quad \to \quad (\lambda x.x\ (z\ x))\ z$$

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

Exercise: equivalence of the two formalizations (first way)

Prove that if

$$t \to_\beta t'$$

then there are $\mathcal{C}$, x, u, v such that

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

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 »

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

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

New base reduction rules

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

New congruence rules

$$\frac{t_1 \quad \to \quad t_1'}{t_1 \oplus t_2 \quad \to \quad t_1' \oplus t_2} \qquad \frac{t_2 \quad \to \quad t_2'}{t_1 \oplus t_2 \quad \to \quad t_1 \oplus t_2'}$$

Extended definitions

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

Booleans and conditionals

New shapes of terms

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

New base rules

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

+ new congruence rules

Pairs

New shapes of terms

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

New base rules

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

+ 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

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

+ congruence rules

Recursion

New shapes of terms

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

New base rules

$$\mathsf{Fix}(t) \quad \rightarrow \quad t\ (\mathsf{Fix}(t))$$

+ congruence rules

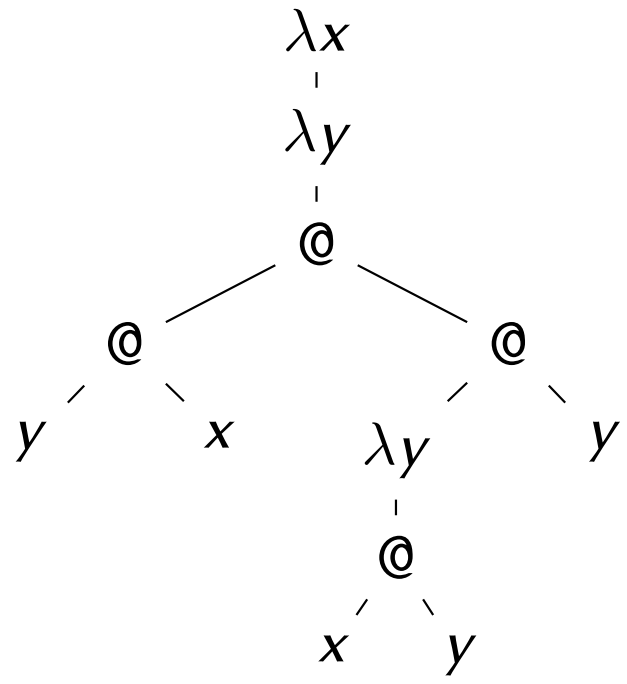
Exercise : extended reduction

Compute the value of the expression

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

Use numbers instead of variable names

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



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

$$\lambda.\lambda.0\ 1\ ((\lambda.2\ 0)\ 0)$$

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)

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

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

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

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

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

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. $\lambda x.(\lambda x.x\ y)(\lambda y.x\ y)$
2. $\lambda x\ y.x(\lambda y.(\lambda y.y)y\ z)$

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

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

Homework

Write the proof down carefully and submit it before next course

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

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

Find (simple) counter-examples when one hypothesis or the other is not satisfied