CSC_51051_EP: Pure λ-calculus

Samuel Mimram

2025

École polytechnique

Part I

Introduction

Imperative programming

You are mostly used to **imperative** programming languages where programs consist in sequences of instructions and modify a state.

```
public long factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}
```

Functional programming

In **functional** programming, we manipulate functions, which can even be created on the fly:

```
let rec map f l =
  match l with
  | [] -> []
  | x::l' -> (f x)::(map f l')

let double_list l =
  map (fun x -> 2 * x) l
```

So that

```
# double_list [1; 2; 3];;
- : int list = [2; 4; 6]
```

Functional programming

We can define the multiplication function by

```
let mult x y = x * y
```

and then define doubling with

```
let double = mult 2
```

this is thanks to Curryfication which allows partial application:
the above definition is equivalent to

```
let mult = fun x -> fun y -> x * y
```

λ-calculus

We have seen how to describe the reduction for an imperative programming language.

How can we define this for functional programming languages?

The λ-**calculus** is the core of a functional programming language:
we focus on the functional part.

It is a subject of study per se, but it can be mixed with imperative features
(e.g. OCaml).

Variable binding

When we define a function

$$f(x) = 2 \times x$$

the name of the variable x does not matter:

$$f(y) = 2 \times y$$

is considered to be the same function.

We say that x is **bound** in the expression.

The relation which identifies two expressions differing only in renaming of bound variable is called α-**conversion**.

Variable binding

There are many places where this phenomenon occurs in mathematics:

$$\lim_{x \to \infty} \frac{y}{x} \qquad \int_0^1 tx \, \mathrm{d}t \qquad \sum_{i=0}^{n} ix$$

Variable binding

This looks like a detail, but it is quite important: consider

$$f(y) = \lim_{x \to \infty} \frac{y}{x}$$

Clearly, if I replace y by any arbitrary expression t (say, $t = \ln(sin(z))^{\sqrt{2}}$),

$$f(t) = \lim_{x \to \infty} \frac{t}{x} = 0$$

But what about $y = x$?

$$f(x) = \lim_{x \to \infty} \frac{x}{x} = \lim_{x \to \infty} 1 = 1$$

We always implicitly make the assumption that bounded variables are **fresh**, i.e. do not occur in substituted terms, which we can do up to α-conversion:

$$f(x) = \left(y \mapsto \lim_{x \to \infty} \frac{y}{x}\right)(x) = \left(y \mapsto \lim_{z \to \infty} \frac{y}{z}\right)(x) = \lim_{z \to \infty} \frac{x}{z} = 0$$

Variable binding

In mathematics, this is generally implicit, but when implementing we have to explicitly take care of α-**conversion**: there is no easy way of automatically taking care of this.

Believe it or not, this is one of the most error prone issues to correctly handle.

The λ notation

Instead of the mathematical notation

$$x \mapsto t$$

or the programming notation

```
fun x -> t
```

we write

$$\lambda x.t$$

where x might occur in the term t, e.g.

$$\lambda x.(2 \times x)$$

Moreover, we will always write

$$f = \lambda x.t \qquad \text{instead of} \qquad f(x) = t$$

The "squaring" function can be defined as

$$\text{square} = \lambda x.(x \times x)$$

We can then apply the function to an argument

$$\text{square } 3$$

which will reduce to

$$3 \times 3$$

as expected.

We can also consider the function

$$\text{mult} \quad = \quad \lambda x.\lambda y.(x \times y)$$

We expect $\text{mult}\ t$ to be multiplication by t.

We should not have

$$\text{mult}\ y \quad \longrightarrow \quad \lambda y.(y \times y)$$

but

$$\text{mult}\ y = (\lambda x.\lambda y.(x \times y))y = (\lambda x.\lambda z.(x \times z))y \quad \longrightarrow \quad \lambda z.(y \times z)$$

Part II

λ-calculus

λ-calculus

This notation was invented by Church in the 1930s, looking for new foundations of mathematics based on functions instead of sets.

The set of λ-**terms** is defined by the following grammar:

$$t, u ::= x \mid t\ u \mid \lambda x.t$$

A λ-term is thus either

- a *variable* x,
- an *application* $t\ u$,
- an *abstraction* $\lambda x.t$.

For instance,

$$\lambda x.x \qquad (\lambda x.(xx))(\lambda y.(yx)) \qquad \lambda x.(\lambda y.(x(\lambda z.y)))$$

Conventions

By convention,

- application is associative on the left, i.e.

$$tuv = (tu)v$$

  and not $t(uv)$,
- application binds more tightly than abstraction, i.e.

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

  and not $(\lambda x.x)y$ (this says that abstraction extends as far as possible on the right),
- we sometimes group abstractions, i.e.

$$\lambda xyz.xz(yz) \qquad \text{is read as} \qquad \lambda x.\lambda y.\lambda z.xz(yz)$$

Bound and free variables

We write $\mathsf{FV}(t)$ for the set of **free variables** of t, i.e. those which are not bound by a λ.

For instance,

$$\mathsf{FV}(\lambda x.x\,y\,z) = \{y, z\} \qquad \mathsf{FV}((\lambda x.x)x) = \{x\} \qquad \mathsf{FV}((\lambda x.x)(\lambda y.y)) = \emptyset$$

Formally,

$$\begin{aligned} \mathsf{FV}(x) &= \{x\} \\ \mathsf{FV}(t\,u) &= \mathsf{FV}(t) \cup \mathsf{FV}(u) \\ \mathsf{FV}(\lambda x.t) &= \mathsf{FV}(t) \setminus \{x\} \end{aligned}$$

α-equivalence

Two terms are α-**equivalent** when they only differ by renaming of bound variables.

In a subterm, of the form $\lambda x.t$, we can rename x to y only if $y \notin \mathsf{FV}(t)$.

For instance,

$$(\lambda x.xxy)t =\!=_\alpha (\lambda z.zzy)t \neq\!=_\alpha (\lambda y.yyy)t$$

In the following terms are always considered up to α-equivalence.

Substitution

We write $t[u/x]$ for the term t where all *free* occurrences of x have been replaced by u.

$$
\begin{aligned}
x[u/x] &= u \\
y[u/x] &= y && \text{if } y \neq x \\
(t_1\, t_2)[u/x] &= (t_1[u/x])\,(t_2[u/x]) \\
(\lambda x.t)[u/x] &= \lambda x.t && \text{(simple but useful optimization)} \\
(\lambda y.t)[u/x] &= \lambda y.(t[u/x]) && \underline{\text{if } y \neq x \text{ and } y \notin \mathrm{FV}(u)}
\end{aligned}
$$

For instance,

$$(\lambda x.x)[y/x] \quad \stackrel{\alpha}{=} \quad (\lambda z.z)[y/x] \quad = \quad \lambda z.z \quad \stackrel{\alpha}{=} \quad \lambda x.x$$

β-reduction

The notion of "execution" for λ-terms is given by β-**reduction**.

A β-**reduction step** consists in replacing a subterm

$$(\lambda x.t)\, u \quad \longrightarrow_\beta \quad t[u/x]$$

Such a subterm is called a β-**redex**.

For instance,

$$\begin{aligned}(\lambda x.y)(\underline{(\lambda z.zz)(\lambda t.t)}) &\longrightarrow_\beta (\lambda x.y)(\underline{(\lambda t.t)(\lambda t.t)}) \\ &\longrightarrow_\beta \underline{(\lambda x.y)(\lambda t.t)} \\ &\longrightarrow_\beta y\end{aligned}$$

β-reduction

- Reduction can create β-redexes:

$$(\lambda x.xx)(\lambda y.y) \longrightarrow_\beta (\lambda y.y)(\lambda y.y)$$

- Reduction can duplicate β-redexes:

$$(\lambda x.xx)((\lambda y.y)(\lambda z.z)) \longrightarrow_\beta ((\lambda y.y)(\lambda z.z))((\lambda y.y)(\lambda z.z))$$

- Reduction can erase β-redexes:

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

β-reduction

- Some terms cannot reduce, **normal forms**:

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

- Some terms reduce infinitely:

$$(\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta (\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta \dots$$

- Some terms reduce in multiple ways:

$$\lambda y.y \; {}_\beta\!\longleftarrow (\lambda xy.y)((\lambda x.x)(\lambda x.x)) \longrightarrow_\beta (\lambda xy.y)(\lambda x.x)$$

β-reduction

A β-**reduction path** is a sequence of β-reduction steps:

$$t \xrightarrow{*}_\beta u \qquad = \qquad t \longrightarrow_\beta t_1 \longrightarrow_\beta t_2 \longrightarrow_\beta \ldots \longrightarrow_\beta u$$

(by which we mean that there exists terms t_i with the above reductions)

The number of β-reduction steps is called the **length** of the path.

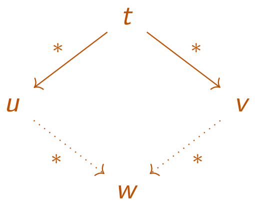
A reasonable programming language should be “deterministic” or at least “reasonably predictable”.

How can we formalize this property?

Confluence

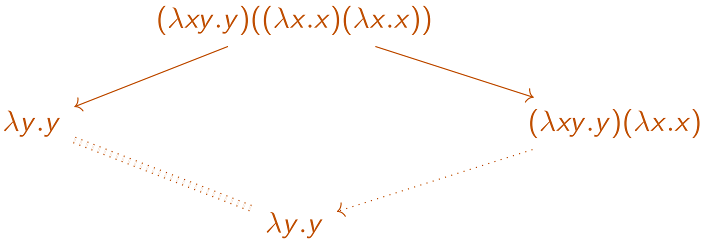
A fundamental property of β-reduction is that we can always make two reductions from the same term converge.

Theorem (Confluence)
Given a term t such that $t \xrightarrow{*}_\beta u$ and $t \xrightarrow{*}_\beta v$ there exists a term w such that $u \xrightarrow{*}_\beta w$ and $v \xrightarrow{*}_\beta w$:



Confluence

For instance,



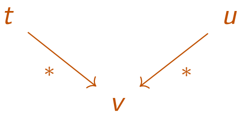
The β-**equivalence** $=_\beta$ is the smallest equivalence relation containing $\longrightarrow_\beta$.

Two terms t and u are β-**equivalent** if there exists a sequence of reductions

$$t =_\beta u \qquad = \qquad t \overset{*}{\longleftarrow} t_1 \overset{*}{\longrightarrow} t_2 \overset{*}{\longleftarrow} t_3 \overset{*}{\longrightarrow} t_4 \overset{*}{\longleftarrow} \dots \overset{*}{\longrightarrow} u$$

From confluence,

Theorem (Church-Rosser)
Two terms t and u are β-equivalent iff there exists v such that $t \overset{*}{\longrightarrow}_\beta v$ and $u \overset{*}{\longrightarrow}_\beta v$:



Another equivalence

This is not the only interesting notion of equivalence.

The η-equivalence $=_\eta$ is the smallest congruence such that, for every term t,

$$t \quad =_\eta \quad \lambda x.t\,x$$

when $x \notin \mathsf{FV}(t)$.

For instance, in OCaml

$$\texttt{sin} \quad =_\eta \quad \texttt{fun x -> sin x}$$

We will not insist much on it in the following, but we will see that two such functions can behave differently in languages such as OCaml (but not in λ-calculus).

Poll

How difficult is it do decide whether two terms are β-equivalent?

Part III

Expressive power

Let's see what we can compute
within
the pure λ-calculus.

Identity

We define the **identity** by

$$\mathsf{I} = \lambda x.x$$

It satisfies

$$\mathsf{I}\, t \longrightarrow_{\beta} t$$

Booleans

The **booleans** can be encoded as the two projections

$$\mathsf{T} = \lambda xy.x \qquad\qquad \mathsf{F} = \lambda xy.y$$

Conditional branching can be encoded as

$$\mathsf{if} = \lambda bxy.b\,x\,y$$

Namely,

$$\mathsf{if}\ \mathsf{T}\ t\ u \xrightarrow{*}_{\beta} t \qquad\qquad \mathsf{if}\ \mathsf{F}\ t\ u \xrightarrow{*}_{\beta} u$$

For instance, the first reduction is

$$\begin{aligned}\mathsf{if}\ \mathsf{T}\ t\ u = (\lambda bxy.bxy)(\lambda xy.x)tu &\longrightarrow_{\beta} (\lambda xy.(\lambda xy.x)xy)tu\\ &\longrightarrow_{\beta} (\lambda y.(\lambda xy.x)ty)u\\ &\longrightarrow_{\beta} (\lambda xy.x)tu\\ &\longrightarrow_{\beta} (\lambda y.t)u \longrightarrow_{\beta} t\end{aligned}$$

Booleans

We can the implement usual boolean operations:

$$\begin{aligned} \text{and} &= \lambda xy.\text{if } x\, y\, \mathsf{F} \\ &= \lambda xy.x\, y\, \mathsf{F} \end{aligned} \qquad \begin{aligned} \text{or} &= \lambda xy.\text{if } x\, \mathsf{T}\, y \\ &= \lambda xy.x\, \mathsf{T}\, y \end{aligned} \qquad \begin{aligned} \text{not} &= \lambda x.\text{if } x \mathsf{F}\, \mathsf{T} \\ &= \lambda xy.x\, \mathsf{F}\, \mathsf{T} \end{aligned}$$

There are other possible implementations, e.g.

$$\text{and} = \lambda xy.x\, y\, x$$

(not β-equivalent, note that behavior is only specified on booleans)

Pairs

We can encode pairs from booleans:

$$\mathsf{pair} = \lambda xyb.\mathsf{if}\ b\ x\ y$$

Namely,

$$\mathsf{pair}\ t\ u \xrightarrow{*}_\beta \lambda b.\mathsf{if}\ b\ t\ u$$

and we have

$$(\mathsf{pair}\ t\ u)\ \mathsf{T} \xrightarrow{*}_\beta t \qquad\qquad (\mathsf{pair}\ t\ u)\ \mathsf{F} \xrightarrow{*}_\beta u$$

We can thus define

$$\mathsf{fst} = \lambda p.p\ \mathsf{T} \qquad\qquad \mathsf{snd} = \lambda p.p\ \mathsf{F}$$

which behaves as expected

$$\mathsf{fst}\ (\mathsf{pair}\ t\ u) \xrightarrow{*}_\beta t \qquad\qquad \mathsf{snd}\ (\mathsf{pair}\ t\ u) \xrightarrow{*}_\beta u$$

It is not much more difficult to encode tuples.

Natural numbers

The n-th **Church numeral** is the λ-term

$$\underline{n} = \lambda fx.f^n x = \lambda fx.f(f(\dots(fx)))$$

so that

$$\underline{0} = \lambda fx.x \qquad \underline{1} = \lambda fx.fx \qquad \underline{2} = \lambda fx.f(fx) \qquad \underline{3} = \lambda fx.f(f(fx)) \qquad \dots$$

We can program successor as

$$\mathrm{succ} = \lambda nfx.f(nfx)$$

and other arithmetical operations:

$$\mathrm{add} = \lambda mnfx.mf(nfx) \qquad \mathrm{mul} = \lambda mnfx.m(nf)x \qquad \mathrm{exp} = \lambda mn.nm$$

and the test at zero:

$$\mathrm{iszero} = \lambda n.n(\lambda z.\mathrm{F})\mathrm{T}$$

Natural numbers

We can also program the predecessor

$$\text{pred} = \lambda nfx.n(\lambda gh.h(gf))(\lambda y.x)(\lambda y.y)$$

(see in TD) and thus subtraction by

$$\text{sub} = \lambda mn.n\,\text{pred}\,m$$

Fixpoints

In order to be able to program more full-fledged programs, we need to be able to define recursive functions.

For instance,

```
let rec fact n =
  if n = 0 then 1 else n * fact (n-1)
```

Fixpoints

In mathematics, a **fixpoint** of a function $f : A \to A$ is an element $a \in A$ such that

$$f(a) = a$$

A distinguishing feature of λ-calculus is that

- every program admits a fixpoint,
- this fixpoint can be computed within λ-calculus.

This means that there is a term Y such that

$$t\,(\mathsf{Y}\,t) =_\beta \mathsf{Y}\,t$$

This can be used to program recursive functions!

How do we program a fixpoint operator in OCaml?

$$\texttt{fix}\,t \quad = \quad t\,(\texttt{fix}\,t)$$

Fixpoints

In OCaml, we can program a fixpoint operator with (by definition)

```
let rec fix f x = f (fix f) x
```

The factorial can then be programmed with

```
let fact_fun f n =
  if n = 0 then 1 else n * f     (n - 1)
```

and then

```
let fact = fix fact_fun
```

Problem solved:

```
# fact 5;;
- : int = 120
```

(by an η-expansion!...)

Fixpoints

This translates directly as

$$\mathsf{fact} = \mathsf{Y}(\lambda fn.\mathsf{if}\,(\mathsf{iszero}\,n)\,\underline{1}\,(\mathsf{mul}\,n\,(f\,(\mathsf{pred}\,n))))$$

The factorial of 2 computes as

$$\begin{aligned}
\mathsf{fact}\,\underline{2} &= (\mathsf{Y}F)\,\underline{2} \\
&\xrightarrow{*}_\beta F\,(\mathsf{Y}F)\,\underline{2} \\
&\xrightarrow{*}_\beta \mathsf{if}\,(\mathsf{iszero}\,\underline{2})\,\underline{1}\,(\mathsf{mul}\,\underline{2}\,((\mathsf{Y}F)\,(\mathsf{pred}\,\underline{2}))) \\
&\xrightarrow{*}_\beta \mathsf{if}\,\mathsf{false}\,\underline{1}\,(\mathsf{mul}\,\underline{2}\,((\mathsf{Y}F)\,(\mathsf{pred}\,\underline{2}))) \\
&\xrightarrow{*}_\beta \mathsf{mul}\,\underline{2}\,((\mathsf{Y}F)\,(\mathsf{pred}\,\underline{2})) \\
&\xrightarrow{*}_\beta \mathsf{mul}\,\underline{2}\,((\mathsf{Y}F)\,\underline{1}) \\
&\vdots \\
&\xrightarrow{*}_\beta \mathsf{mul}\,\underline{2}\,(\mathsf{mul}\,\underline{1}\,\underline{1}) \xrightarrow{*}_\beta \underline{2}
\end{aligned}$$

Fixpoints

```
(((λf.((λx.(f (x x))) (λx.(f (x x))))) (λf.(λn.((((λb.(λx.(λy.((b x) y)))) ((λn.(λx.(λy.((n (λz.y)) x)))) n))
↪  (λf.(λx.(f x)))) (((λm.(λn.(λf.(λx.((m (n f)) x))))) n) (f ((λn.((λp.(p (λx.(λy.x)))) ((n
↪  (λp.(((λx.(λy.(λb.((((λb.(λx.(λy.((b x) y)))) b) x) y)))) ((λp.(p (λx.(λy.y)))) p)) ((λn.(λf.(λx.((n f) (f x)))))
↪  ((λp.(p (λx.(λy.y)))) p))))) (((λx.(λy.(λb.((((λb.(λx.(λy.((b x) y)))) b) x) y)))) (λf.(λx.x))) (λf.(λx.x))))))
↪  n))))))) (λf.(λx.(f (f x)))))
-> (((λx.((λf.(λn.((((λb.(λx.(λy.((b x) y)))) ((λn.(λx.(λy.((n (λz.y)) x)))) n)) (λf.(λx.(f x))))
↪  (((λm.(λn.(λf.(λx.((m (n f)) x))))) n) (f ((λn.((λp.(p (λx.(λy.x)))) ((n (λp.(((λx.(λy.(λb.((((λb.(λx.(λy.((b x)
↪  y)))) b) x) y)))) ((λp.(p (λx.(λy.y)))) p)) ((λn.(λf.(λx.((n f) (f x))))) ((λp.(p (λx.(λy.y)))) p)))))
↪  (((λx.(λy.(λb.((((λb.(λx.(λy.((b x) y)))) b) x) y)))) (λf.(λx.x))) (λf.(λx.x)))))) n)))))) (x x)))
↪  (λx.((λf.(λn.((((λb.(λx.(λy.((b x) y)))) ((λn.(λx.(λy.((n (λz.y)) x)))) n)) (λf.(λx.(f x)))) (((λm.(λn.(λf.(λx.((m
↪  (n f)) x))))) n) (f ((λn.((λp.(p (λx.(λy.x)))) ((n (λp.(((λx.(λy.(λb.((((λb.(λx.(λy.((b x) y)))) b) x) y)))) ((λp.(p
↪  (λx.(λy.y)))) p)) ((λn.(λf.(λx.((n f) (f x))))) ((λp.(p (λx.(λy.y)))) p))))) (((λx.(λy.(λb.((((λb.(λx.(λy.((b x)
↪  y)))) b) x) y)))) (λf.(λx.x))) (λf.(λx.x)))))) n)))))) (x x)))) (λf.(λx.(f (f x)))))
.
.
.
-> (λf.(λx.(f ((((λx.x) (λf.(λx.(f x)))) f) x))))
-> (λf.(λx.(f (((λf.(λx.(f x))) f) x))))
-> (λf.(λx.(f ((λx.(f x)) x))))
-> (λf.(λx.(f (f x))))
333 steps
```

Fixpoints

We can also write unbounded loops:

```
let min_from_fun f p n =
  if p n then n else f p (n+1)

let min_from = fix min_from_fun

let min p = min_from p 0

let x = min (fun n -> n - 10 = 0)
```

Fixpoints

We thus have

- natural numbers,
- the successor function,
- tuples and projections,
- composition,
- conditional branching with test to zero,
- recursion.

We thus have **recursive functions**!

Turing completeness

This should convince you that the λ-calculus is **Turing complete**.

Theorem
The following decision problems are undecidable:

- *whether two λ-terms are β-equivalent,*
- *whether a λ-term can reduce to a normal form.*

Fixpoints

...excepting that we have not explained how to define a **fixpoint combinator** Y yet.

The OCaml implementation

```
let rec fix f x = f (fix f) x
```

does not translate to λ-calculus because it is not an anonymous function:

```
let fix = fun f -> ???
```

Any guess?

Fixpoints

We can start by recalling that we had a non-terminating term:

$$\Omega = (\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta (\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta \ldots$$

We can obtain the fixpoint combinator by a slight modification:

$$\mathsf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Namely,

$$\mathsf{Y}\, f \longrightarrow (\lambda x.f(xx))(\lambda x.f(xx)) \longrightarrow \underbrace{f\,((\lambda x.f(xx))(\lambda x.f(xx)))}_{f\,(\mathsf{Y}\, f)} \longrightarrow \ldots$$

i.e.

$$\mathsf{Y}\, f =_\beta f\,(\mathsf{Y}\, f)$$

Fixpoints

Note that computing fixpoints can loop:

$$\mathsf{Y}\, f \xrightarrow{*}_\beta f\,(\mathsf{Y}\, f) \xrightarrow{*}_\beta f\,(f\,(\mathsf{Y}\, f)) \xrightarrow{*}_\beta \ldots$$

So that our implementation of factorial can loop
(this is what was happening in OCaml).

However, programming languages implement a **reduction strategy**, i.e. a particular way of β-reducing programs.

If we choose a decent one, the factorial will compute the factorial.

Fixpoints

Does this work in practice (= OCaml)?

```
let fix = fun f -> (fun x -> f (x x)) (fun x -> f (x x))
```

```
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
       The type variable 'a occurs inside 'a -> 'b
```

Namely, `x x` means that

- `x` is a function: of type `'a -> 'b`,
- that `'a` = `'a -> 'b`

i.e. the type of `x` should be

```
... -> 'b -> 'b -> 'b -> 'b
```

Fixpoints

There are ways to get around this, one being to use the option `-rectypes` of OCaml (which allows types such as `('a -> 'b) as 'a`):

```
let fix = fun f -> (fun x y -> f (x x) y) (fun x y -> f (x x) y)
```

has type

```
(('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

and we define

```
let fact_fun f n = if n = 0 then 1 else n * f (n - 1)
let fact = fix fact_fun
```

Problem solved:

```
# fact 5;;
- : int = 120
```

Fixpoints

If you (understandably) don't feel comfortable with `-rectypes`:

```
type 'a t = Arr of ('a t -> 'a)

let arr (Arr f) = f

let fix = fun f -> (fun x y -> f (arr x x) y)
                   (Arr (fun x y -> f (arr x x) y))

let fact_fun f n = if n = 0 then 1 else n * f (n - 1)

let fact = fix fact_fun

let n = fact 5
```

More primitives: products

In practice (= OCaml), one does not encode everything in *pure* λ-calculus, but rather adds more primitives. For instance, **products** can be added with

$$t, u ::= x \mid t\, u \mid \lambda x.t \mid \langle t, u \rangle \mid \pi_\mathsf{l} \mid \pi_\mathsf{r}$$

with additional reduction rules

$$\pi_\mathsf{l}\, \langle t, u \rangle \longrightarrow_\beta t \qquad\qquad \pi_\mathsf{r}\, \langle t, u \rangle \longrightarrow_\beta u$$

and similarly for other constructions.

Reduction strategies

We have seen that the way reduction is implemented has an influence.

The main choice roughly is, for

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

to either

- reduce u to $\hat{u}$ and then reduce $t[\hat{u}/x]$ (*call-by-value*):
  more efficient since we compute arguments once,
- reduce $t[u/x]$ (*call-by-name*):
  not sensitive to divergence of arguments, e.g. $(\lambda xy.y)\Omega\mathrm{I}$.

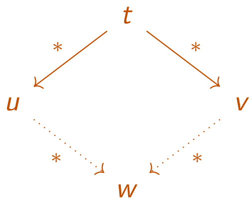
Part IV

Confluence

Confluence

We have announced the confluence theorem:

Theorem (Confluence)
Given a term t such that $t \xrightarrow{*}_\beta u$ and $t \xrightarrow{*}_\beta v$
there exists a term w such that $u \xrightarrow{*}_\beta w$ and $v \xrightarrow{*}_\beta w$:

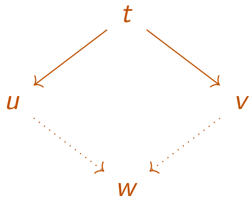


What could be a proof strategy to show confluence?

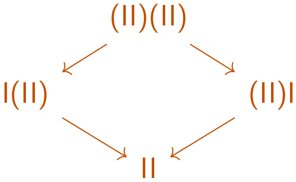
(clearly, we cannot consider all coinitial pairs of reduction paths)

Showing confluence: the diamond property

Maybe we can show that has the **diamond property**:

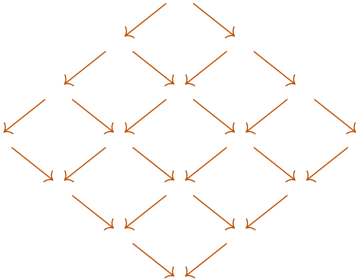


For instance,



Showing confluence: the diamond property

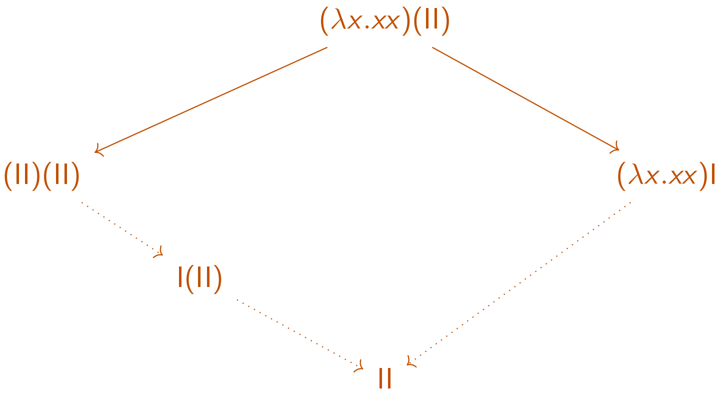
We can then easily conclude to confluence:



Note that this is done by using two recurrences.

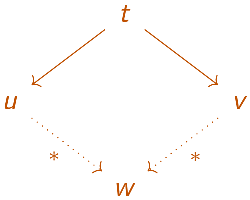
Showing confluence: the diamond property

Excepting that λ-calculus does not satisfy the diamond property:



Showing confluence: local confluence

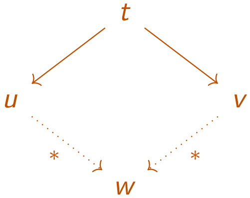
By case analysis, we can show **local confluence**:



from which we cannot deduce confluence. Why?

Showing confluence: local confluence

By case analysis, we can show **local confluence**:



but this does not imply confluence.

Namely, the following situation

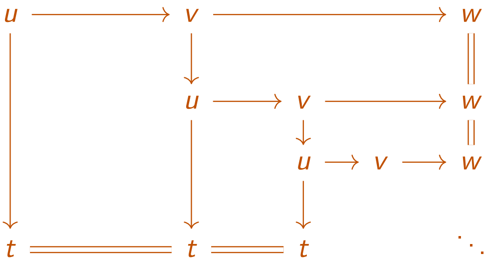
is locally confluent but not confluent.

Showing confluence: local confluence

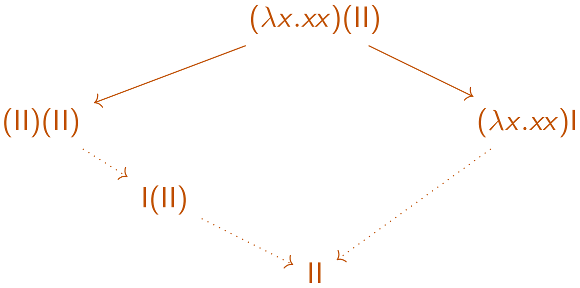
With

$$t \longleftarrow u \rightleftarrows v \longrightarrow w$$

we have



The idea is to use an auxiliary reduction, which does have the diamond property and whose confluence implies the one of β-reduction.



The β-reduction

The β-**reduction** consists in replacing a subterm

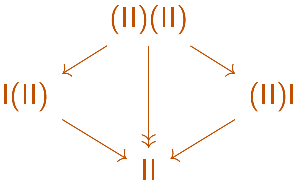
$$(\lambda x.t)\, u \quad \longrightarrow_\beta \quad t[u/x]$$

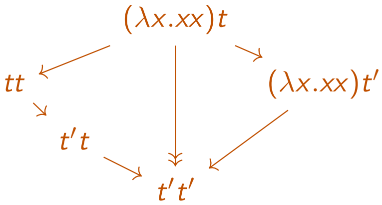
This thus is the smallest relation such that

$$\frac{}{(\lambda x.t)u \longrightarrow_\beta t[u/x]} \qquad \frac{t \longrightarrow_\beta t'}{\lambda x.t \longrightarrow_\beta \lambda x.t'} \qquad \frac{t \longrightarrow_\beta t'}{tu \longrightarrow_\beta t'u} \qquad \frac{u \longrightarrow_\beta u'}{tu \longrightarrow_\beta tu'}$$

The parallel β-reduction

We would like to allow multiple reductions in parallel, e.g.





We define the **parallel β-reduction** as

$$\frac{}{x \twoheadrightarrow x} \qquad \frac{t \twoheadrightarrow t' \qquad u \twoheadrightarrow u'}{(\lambda x.t)u \twoheadrightarrow t'[u'/x]} \qquad \frac{t \twoheadrightarrow t' \qquad u \twoheadrightarrow u'}{t\,u \twoheadrightarrow t'u'} \qquad \frac{t \twoheadrightarrow t'}{\lambda x.t \twoheadrightarrow \lambda x.t'}$$

The parallel β-reduction

The parallel β-reduction thus allows to perform multiple reductions in parallel:

Lemma
If $t \longrightarrow\!\!\!\!\rightarrow u$ then $t \xrightarrow{*}_\beta u$.

Conversely, any β-reduction step is in particular, one β-reduction step in "parallel":

Lemma
If $t \longrightarrow_\beta u$ then $t \longrightarrow\!\!\!\!\rightarrow u$.

Therefore,

Lemma
We have $t \xrightarrow{*}_\beta u$ iff $t \overset{*}{\longrightarrow\!\!\!\!\rightarrow} u$.

Proof.

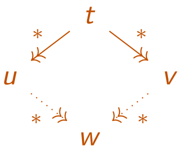
$$\begin{aligned} t \overset{*}{\longrightarrow\!\!\!\!\rightarrow} u &= t \longrightarrow\!\!\!\!\rightarrow t_1 \longrightarrow\!\!\!\!\rightarrow t_2 \longrightarrow\!\!\!\!\rightarrow \dots \longrightarrow\!\!\!\!\rightarrow u \\ &= t \xrightarrow{*}_\beta t_1 \xrightarrow{*}_\beta t_2 \xrightarrow{*}_\beta \dots \xrightarrow{*}_\beta u \end{aligned}$$

□

Our goal is to show:

Theorem

The parallel β-reduction is confluent: if $t \overset{*}{\twoheadrightarrow} u$ and $t \overset{*}{\twoheadrightarrow} v$ then there exists w such that $u \overset{*}{\twoheadrightarrow} w$ and $u \overset{*}{\twoheadrightarrow} w$.



Corollary

The β-reduction is confluent.

We first need some lemmas.

Parallel β-reduction: reflexivity

Lemma
For every term t, we have $t \longrightarrow\!\!\!\!\!\rightarrow t$.

Proof.
By induction on the term t:

$$\frac{}{x \twoheadrightarrow x} \qquad \frac{t \twoheadrightarrow t' \qquad u \twoheadrightarrow u'}{(\lambda x.t)u \twoheadrightarrow t'[u'/x]} \qquad \frac{t \twoheadrightarrow t' \qquad u \twoheadrightarrow u'}{t\,u \twoheadrightarrow t'\,u'} \qquad \frac{t \twoheadrightarrow t'}{\lambda x.t \twoheadrightarrow \lambda x.t'}$$

$\square$

Parallel β-reduction and substitution

Lemma
If $t \longrightarrow\!\!\!\!\!\rightarrow t'$ *and* $u \longrightarrow\!\!\!\!\!\rightarrow u'$ *then* $t[u/x] \longrightarrow\!\!\!\!\!\rightarrow t'[u'/x]$.

Proof.
By induction on the derivation of $t \longrightarrow\!\!\!\!\!\rightarrow t'$.

If $t = \lambda y.t_1$ with $y \neq x$ and we used

$$\frac{t_1 \longrightarrow\!\!\!\!\!\rightarrow t_1'}{\lambda y.t_1 \longrightarrow\!\!\!\!\!\rightarrow \lambda y.t_1'}$$

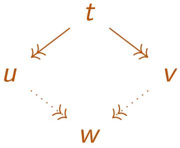
we have $t[u/x] = \lambda y.t_1[u/x] \longrightarrow\!\!\!\!\!\rightarrow \lambda y.t_1'[u'/x] = t'[u'/x]$
since, using the induction hypothesis,

$$\frac{t_1[u/x] \longrightarrow\!\!\!\!\!\rightarrow t_1'[u'/x]}{\lambda y.t_1[u/x] \longrightarrow\!\!\!\!\!\rightarrow \lambda y.t_1'[u'/x]}$$

□

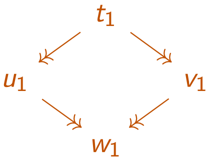
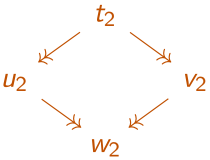
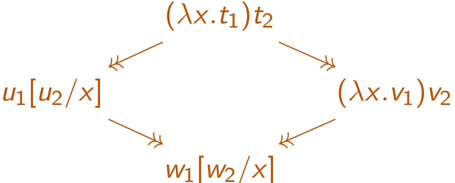
Parallel β-reduction: diamond property

Theorem
The parallel β-reduction has the ***diamond property****: if $t \longrightarrow\!\!\!\!\!\!\rightarrow u$ and $t \longrightarrow\!\!\!\!\!\!\rightarrow v$ then there exists w such that $u \longrightarrow\!\!\!\!\!\!\rightarrow w$ and $v \longrightarrow\!\!\!\!\!\!\rightarrow w$.*



Proof.
By induction on the derivation of $t \longrightarrow\!\!\!\!\!\!\rightarrow u$.

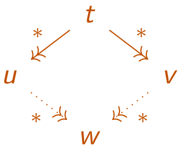
If $\dfrac{t_1 \twoheadrightarrow u_1 \qquad t_2 \twoheadrightarrow u_2}{(\lambda x.t_1)t_2 \twoheadrightarrow u_1[u_2/x]}$ and $\dfrac{\lambda x.t_1 \twoheadrightarrow \lambda x.v_1 \qquad t_2 \twoheadrightarrow v_2}{(\lambda x.t_1)\, t_2 \twoheadrightarrow (\lambda x.v_1)\, v_2}$ then


and

thus

.

Parallel β-reduction: confluence

Theorem

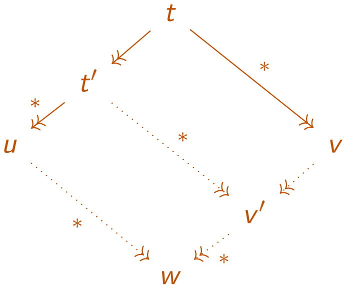
The parallel β-reduction is confluent: if $t \xrightarrow{*} u$ and $t \xrightarrow{*} v$ then there exists w such that $u \xrightarrow{*} w$ and $u \xrightarrow{*} w$.



Proof.

By induction on the length of the reduction $t \xrightarrow{*} u$.

- Otherwise,



□

Part V

De Bruijn indices

α-conversion, again

Again, α-conversion (renaming of bound variables) is one of the greatest source of bugs and problems.

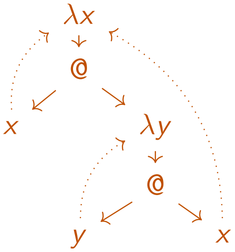
An idea to eliminate the need for renaming is consists in having a convention for naming variables.

De Bruijn indices

In a closed term, such as

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

every variable is bound by some λ-abstract above:



The **de Bruijn convention**: replace every variable by the number of λs to jump over

$$\lambda.0(\lambda.01)$$

De Bruijn indices

We now consider λ-terms generated by the grammar

$$t, u ::= i \mid t\, u \mid \lambda.t$$

where $i \in \mathbb{N}$ is a **de Bruijn index**.

Again, an index i means the variable declared by the i-th λ above.

If there are not enough λs, then it is a free variable:

$$\lambda x.x x_0 x_2 \qquad \text{becomes} \qquad \lambda.013$$

(we can assume that the free variables are $\{x_0, \ldots, x_{k-1}\}$)

Reduction

The rule for β-reduction is the usual one:

$$(\lambda.t)u \longrightarrow_\beta t[u/0]$$

excepting that the substitution now has to take care of properly handling indices.

Reduction

The reduction

$$\lambda x.(\lambda y.\lambda z.y)\,(\lambda t.t) \longrightarrow_\beta \lambda x.(\lambda z.y)[\lambda t.t/y] = \lambda x.\lambda z.y[\lambda t.t/y] = \lambda x.\lambda z.\lambda t.t$$

corresponds to

$$\lambda.(\lambda.\lambda.1)\,\lambda.0 \longrightarrow_\beta \lambda.(\lambda.1)[\lambda.0/0] = \lambda.\lambda.1[\lambda.0/1] = \lambda.\lambda.\lambda.0$$

and we are tempted to define substitution by

$$\begin{aligned} i[u/i] &= u \\ j[u/i] &= j && \text{for } j \neq i \\ (t\,t')[u/i] &= (t[u/i])\,(t'[u/i]) \\ (\lambda.t)[u/i] &= \lambda.t[u/i{+}1] \end{aligned}$$

Incorrect: in the last case, t might contain free variables.

Reduction

The reduction

$$\lambda x.(\lambda y.\lambda z.y)\, x \longrightarrow_\beta \lambda x.(\lambda z.y)[x/y] = \lambda x.\lambda z.y[x/y] = \lambda x.\lambda z.x$$

corresponds to

$$\lambda.(\lambda.\lambda.1)\, 0 \longrightarrow_\beta \lambda.(\lambda.1)[0/0] = \lambda.\lambda.1[1/1] = \lambda.\lambda.1$$

and the last case of substitution should actually be

$$(\lambda.t)[u/i] = \lambda.t[\uparrow_0 u/i{+}1]$$

where $\uparrow_0 u$ is u with all free variables increased by 1 (and other unchanged).

Still incorrect: β-reduction removes abstractions!

Reduction

The reduction

$$\lambda x.(\lambda y.x)\,(\lambda t.t) \longrightarrow_\beta \lambda x.x[\lambda t.t/y] = \lambda x.x$$

corresponds to

$$\lambda.(\lambda.1)\,(\lambda.0) \longrightarrow_\beta \lambda.1[\lambda.0/0] = 0$$

and the first case of substitution should actually be, for $j \neq i$,

$$j[u/i] = \downarrow_i j$$

with

$$\downarrow_l i = \begin{cases} i - 1 & \text{if } i > l \\ i & \text{if } i < l \end{cases}$$

Reduction

In summary, the β-**reduction** can be defined as

$$(\lambda.t)u \longrightarrow_\beta t[u/0]$$

with

$$\begin{aligned} i[u/i] &= u \\ j[u/i] &= \downarrow_i j && \text{for } j \neq i \\ (t\,t')[u/i] &= (t[u/i])\,(t'[u/i]) \\ (\lambda.t)[u/i] &= \lambda.t[\uparrow_0 u/i{+}1] \end{aligned}$$

Lifting

We are only left to define $\uparrow_0 u$ which is u with all free variables increased by 1.

For instance,

$$\uparrow_0(0(\lambda.01)) = 1(\lambda.02)$$

Note that, under the λ, we should only increase free variables of index $\geqslant 1$.

Lifting

Given a "cutoff level" l, we define

$$\uparrow_l u$$

which is u with all free variables of index $\geqslant l$ increased by 1:

$$\uparrow_l i = \begin{cases} i & \text{if } i < l \\ i+1 & \text{if } i \geqslant l \end{cases}$$

$$\uparrow_l (t\,u) = (\uparrow_l t)\,(\uparrow_l u)$$

$$\uparrow_l (\lambda.t) = \lambda.(\uparrow_{l+1} t)$$

Reduction

We can define a translation from λ-terms to de Bruijn and back.

Theorem
The β-reduction is compatible with translations.

Part VI

Combinatory logic

Combinatory logic

Combinatory logic was introduced by Schönfinkel and Curry, in order to provide a syntax which does not need to use variable binding or α-conversion.

It begins with the observation that all the λ-terms can be generated by composing a finite number of those:

$$\mathsf{S} = \lambda xyz.(xz)(yz) \qquad \mathsf{K} = \lambda xy.x$$

Note that these terms satisfy:

$$\mathsf{S}\; t\; u\; v \longrightarrow_\beta (t\; v)\,(u\; v) \qquad \mathsf{K}\; t\; u \longrightarrow_\beta t$$

Combinatory logic

The terms are defined as

$$T, U ::= x \mid T\,U \mid \mathsf{S} \mid \mathsf{K}$$

where x is a variable.

The reduction rules are

$$\frac{}{\mathsf{S}\,T\,U\,V \longrightarrow (T\,V)\,(U\,V)} \qquad \frac{}{\mathsf{K}\,T\,U \longrightarrow T}$$

$$\frac{T \longrightarrow T'}{T\,U \longrightarrow T'\,U} \qquad \frac{U \longrightarrow U'}{T\,U \longrightarrow T\,U'}$$

For instance,

$$\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,T \longrightarrow (\mathsf{K}\,T)\,(\mathsf{K}\,T) \longrightarrow T$$

Translation to λ-calculus

We define a translation from combinatory terms to λ-terms by

$$[\![x]\!]_\lambda = x \qquad [\![T\,U]\!]_\lambda = [\![T]\!]_\lambda [\![U]\!]_\lambda \qquad [\![\mathsf{K}]\!]_\lambda = \lambda xy.x \qquad [\![\mathsf{S}]\!]_\lambda = \lambda xyz.(xz)(yz)$$

Proposition
Given combinatory terms T and T', we have

$$T \longrightarrow T' \qquad \textit{implies} \qquad [\![T]\!]_\lambda \overset{*}{\longrightarrow}_\beta [\![T']\!]_\lambda$$

Translation from λ-calculus

Given a combinatory term T and a variable x, we define the term $\Lambda x. T$ by

$$\Lambda x.x = \mathsf{I} = \mathsf{S}\,\mathsf{K}\,\mathsf{K}$$

$$\Lambda x. T = \mathsf{K}\ T \qquad \text{if } x \notin \mathrm{FV}(T),$$

$$\Lambda x.(T\ U) = \mathsf{S}\,(\Lambda x. T)\,(\Lambda x. U) \qquad \text{otherwise.}$$

For instance,

$$\begin{aligned}
[\![\lambda x.\lambda y.x]\!]_{\mathrm{cl}} &= \Lambda x.[\![\lambda y.x]\!]_{\mathrm{cl}} \\
&= \Lambda x.\Lambda y.[\![x]\!]_{\mathrm{cl}} \\
&= \Lambda x.\Lambda y.x \\
&= \Lambda x.\mathsf{K}\,x \\
&= \mathsf{S}\,(\Lambda x.\mathsf{K})\,(\Lambda x.x) \\
&= \mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I}
\end{aligned}$$

Properties of the embedding

Lemma
For any λ-term t, $[\![[\![t]\!]_{\mathrm{cl}}]\!]_\lambda \xrightarrow{*}_\beta t$.

For instance,

$$[\![[\![\lambda x.x]\!]_{\mathrm{cl}}]\!]_\lambda = [\![\mathsf{S}\,\mathsf{K}\,\mathsf{K}]\!]_\lambda = (\lambda xyz.(xz)(yz))(\lambda xy.x)(\lambda xy.x) \xrightarrow{*}_\beta \lambda x.x$$

Corollary
Every closed λ-term can be obtained by composing the λ-terms S *and* K.

Limitations of the translations

It is not true that $t \xrightarrow{*}_\beta u$ implies $[\![t]\!]_{\mathrm{cl}} \xrightarrow{*} [\![u]\!]_{\mathrm{cl}}$.

For instance,

$$[\![\lambda x.(\lambda y.y)\, x]\!]_{\mathrm{cl}} = \mathsf{S}\,(\mathsf{K}\,\mathsf{I})\,\mathsf{I} \qquad\qquad [\![\lambda x.x]\!]_{\mathrm{cl}} = \mathsf{I}$$

(both are normal forms!)

However, it gets true if we apply them to enough arguments:

$$\mathsf{S}\,(\mathsf{K}\,\mathsf{I})\,\mathsf{I}\,T \longrightarrow \mathsf{K}\,\mathsf{I}\,T\,(\mathsf{I}\,T) \longrightarrow \mathsf{I}\,(\mathsf{I}\,T) \longrightarrow \mathsf{I}\,T \longrightarrow T \qquad \text{and} \qquad \mathsf{I}\,T \longrightarrow T$$

Limitations of the translation

The translation of a combinatory term in normal form is not necessarily a normal form:

$$[\![\mathsf{K}\,x]\!]_{\mathrm{cl}} = (\lambda xy.x)\,x \longrightarrow_\beta \lambda y.\,x$$

Limitations of the translation

A combinatory term T is not convertible with $[\![[\![T]\!]_\lambda]\!]_{\mathrm{cl}}$ in general

$$[\![[\![\mathsf{K}]\!]_\lambda]\!]_{\mathrm{cl}} = [\![\lambda xy.x]\!]_{\mathrm{cl}} = \mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I} \neq \mathsf{K}$$

(they are both normal forms and combinatory logic can be shown to be confluent)

Limitations of the translation

All those defects are due to the fact that combinatory terms might be stuck (compared to λ-terms) if they don't have enough arguments.

The translation is still quite useful.

Undecidability

The system

$$T, U ::= x \mid T\,U \mid \mathsf{S} \mid \mathsf{K}$$

with rules

$$\frac{}{\mathsf{S}\,T\,U\,V \longrightarrow (T\,V)\,(U\,V)} \qquad \frac{}{\mathsf{K}\,T\,U \longrightarrow T}$$

$$\frac{T \longrightarrow T'}{T\,U \longrightarrow T'\,U} \qquad \frac{U \longrightarrow U'}{T\,U \longrightarrow T\,U'}$$

simulates λ-calculus and is thus undecidable!

Iota

We have reduced λ-calculus to 2 combinators, can we do 1?

Yes,

$$\iota = \lambda x.x \, \mathsf{S} \, \mathsf{K}$$

Namely,

$$\mathsf{I} = \iota \, \iota \qquad \mathsf{K} = \iota \, (\iota \, (\iota \, \iota)) \qquad \mathsf{S} = \iota \, (\iota \, (\iota \, (\iota \, \iota)))$$

The reduction is

$$\iota \, T \longrightarrow T \, (\iota \, (\iota \, (\iota \, (\iota \, \iota)))) \, (\iota \, (\iota \, (\iota \, \iota)))$$

Iota

The terms are generated by the grammar

$$t, u ::= \iota \mid t\, u$$

A term t can be encoded as a binary word $[t]$ defined by

$$[\iota] = 1 \qquad\qquad [t\, u] = 0[t][u]$$

so that $\iota\,(\iota\,(\iota\,\iota))$ is encoded as 0101011.

We thus get an interesting binary encoding of λ-terms.