Computing in the λ-calculus

Samuel Mimram
samuel.mimram@lix.polytechnique.fr
http://lambdacat.mimram.fr

November 16, 2020

We recall that λ-*terms* t are of the form x (a variable) or $\lambda x.t$ (an abstraction) or tu (an application). The β-*reduction* is the closure under context of the relation $(\lambda x.t)u \to t[u/x]$, i.e. the relation generated by

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

We write $\overset{*}{\to}$ (resp. $\overset{*}{\leftrightarrow}$) for the reflexive and transitive (resp. and symmetric) closure of $\to$.

1 Reduction graphs

The *reduction graph* of a λ-term t is the graph, whose vertices are λ-terms, defined as the smallest graph such that t is a vertex and there is an arrow between two vertices t and t' whenever $t \to t'$.

1. Write the respective reduction graphs of $(\lambda x.xx)(\lambda y.y)z$ and $(\lambda xy.x)((\lambda x.xx)(\lambda xy.xy))$.
2. Can a reduction graph have loops? be infinite? be infinitely branching?

   Solution. Yes (take $\Omega = (\lambda x.xx)(\lambda x.xx)$), yes (take $(\lambda x.f(xx))(\lambda x.f(xx))$) and no.

2 Computing in pure λ-calculus

We encode the booleans true and false as the λ-terms

$$\top = \lambda x.\lambda y.x \qquad\qquad \bot = \lambda x.\lambda y.y$$

1. Define a λ-term if encoding conditional branching: we should have

   $$\text{if } \top\, t\, u \overset{*}{\to} t \qquad\qquad \text{if } \bot\, t\, u \overset{*}{\to} u$$

   Solution. We define $\text{if} = \lambda btu.btu$.
2. Define λ-terms encoding conjunction, disjunction and negation of booleans.

   Solution. We define

   $$\text{and} = \lambda ab.\,\text{if}\ a\, b\, \bot \qquad \text{or} = \lambda ab.\,\text{if}\ a\, \top\, b \qquad \text{not} = \lambda a.\,\text{if}\ a\, \bot\, \top$$
3. Define an encoding of pairs of terms in λ-calculus, as well as projections.

   Solution. We define

   $$\text{pair} = \lambda xyb.\,\text{if}\ b\, x\, y \qquad \pi_1 = \lambda p.p\top \qquad \pi_2 = \lambda p.p\bot$$

The Church encoding of a natural number n in λ-calculus is

$$\lambda fx.\underbrace{f(f \ldots (f\, x))}_{n \text{ times}}$$

4. Define the interpretation of the successor, addition, multiplication and exponential functions.

   Solution. We can define

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

   or

   $$\mathrm{add} = \lambda mn.m\,\mathrm{suc}\,n \qquad \mathrm{mul} = \lambda mn.m(\mathrm{add}\,n)0 \quad \mathrm{exp} = \lambda mn.n(\mathrm{mul}\,m)1$$

5. Define a function which tests whether its argument, a natural number, is 0 or not.

   Solution. We define

   $$\mathrm{iszero} = \lambda nxy.n(\lambda z.y)x$$

6. Assuming given the predecessor function, define the subtraction function. Can you see how to define the predecessor?

   Solution. We define

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

A *fixpoint combinator* is a term Y such that

$$\mathrm{Y}\ t \overset{*}{\leftrightarrow} t\,(\mathrm{Y}\ t)$$

7. Recall Russell's paradox in naive set theory.

   Solution. Consider the set $r = \{x \mid x \notin x\}$. If $r \in r$ then $r \notin r$ and if $r \notin r$ then $r \in r$. In other words, $r \in r \Leftrightarrow r \notin r$.

8. Encoding a set t as a predicate which indicates whether an element belongs to it, we can write $t\,u$ instead of $u \in t$, and $\lambda x.t$ instead of $\{x \mid t\}$. Assuming given a term $\neg$ for negation, translate Russell's paradox in λ-calculus, and generalize it in order to obtain a fixpoint combinator Y.

   Solution. We write $r = \lambda x.\neg(xx)$ and we have $rr = \neg(rr)$. Otherwise said, rr is a fixpoint for $\neg$. Generalizing this to any function f instead of $\neg$, we define

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

9. Given a term t, show that the β-equivalence class of Y t is always infinite.

   Solution. We have

   $$\mathrm{Y}\ t \overset{*}{\leftrightarrow} t\,(\mathrm{Y}\ t) \overset{*}{\leftrightarrow} t\,(t\,(\mathrm{Y}\ t)) \overset{*}{\leftrightarrow} \ldots$$

10. Program the factorial function in OCaml. Modify your implementation in order not to use the `rec` keyword, but you can use the function `fix` defined by

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

    In practice, what happens when you evaluate this definition? Fix `fix`.

    Solution. We define the auxiliary function

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

    from which we can deduce the implementation of factorial by

    ```
    let fact = fix fact_fun
    ```

    If we try to evaluate it, we obtain

    ```
    Stack overflow during evaluation (looping recursion?).
    ```

    but we can fix this with an η-expansion of the `fix` function:

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

which is due to the particular evaluation strategy we have in OCaml.

11. Assuming given predecessor, define the factorial function in λ-calculus.

    Solution. We define
    $$\text{fact} = \mathrm{Y}\,(\lambda f n.\,\text{if}\,(\text{iszero}\ n)\,1\,(f\ (\text{pred}\ n)))$$

12. The Fibonacci sequence $(\phi_n)_{n\in\mathbb{N}}$ is defined by $\phi_0 = 0$, $\phi_1 = 1$ and $\phi_n = \phi_{n-1} + \phi_{n+2}$. Give a naive OCaml implementation of this function. What is (roughly) its complexity? Provide a saner implementation.

    Solution. The naive implementation is

```
let rec fib n =
  if n = 0 then 0
  else if n = 1 then 1
  else fib (n-1) + fib (n-2)
```

    whose complexity is exponential. A saner version is obtained by computing two successive values of fib:

```
let fib n =
  let rec aux i (p,q) =
    if i = 0 then (p,q) else aux (i-1) (q,p+q)
  in
  fst (aux n (0,1))
```

13. Implement the predecessor function in OCaml and in λ-calculus.

    Solution. For the predecessor, we can similarly compute the result by iterating n times the function $\phi = (m, n) \mapsto (n, n + 1)$ to $(0, 0)$:

```
let pred n =
  let rec aux i (p,q) =
    if i = 0 then (p,q) else aux (i-1) (q,q+1)
  in
  fst (aux n (0,0))
```

    This easily translates into a λ-term.

14. Show that $\Theta = (\lambda xf.f(xxf))(\lambda xf.f(xxf))$ is also a fixpoint combinator (due to Turing). What is the advantage over Y?

    Solution. We have

$$\begin{aligned}
\Theta t &= (\lambda xf.f(xxf))(\lambda xf.f(xxf))t \\
&\to (\lambda f.f((\lambda xf.f(xxf))(\lambda xf.f(xxf))f))t \\
&\to t((\lambda xf.f(xxf))(\lambda xf.f(xxf))t) \\
&= t(\Theta t)
\end{aligned}$$

    If we look precisely at the situation with Y, we have

$$\begin{aligned}
\mathrm{Y}\,t &= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))t \\
&\to (\lambda x.t(xx))(\lambda x.t(xx)) \\
&\to t((\lambda x.t(xx))(\lambda x.t(xx))) \\
&\leftarrow t(Yt)
\end{aligned}$$

    So the situation is slightly simpler: we have $\Theta t \xrightarrow{*} t(\Theta t)$ as opposed to only $\mathrm{Y}\,t \overset{*}{\leftrightarrow} t(\mathrm{Y}\,t)$.