Computing in the λ-calculus

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

October 17, 2022

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 $\to$.

I 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?

II Booleans

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

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

2. Define λ-terms encoding conjunction, disjunction and negation of booleans.
3. Define an encoding of pairs of terms in λ-calculus, as well as projections.

III Natural numbers

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

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

4. Define the interpretation of the successor, addition, multiplication and exponential functions.
5. Define a function which tests whether its argument, a natural number, is 0 or not.
6. Assuming given the predecessor function (by convention, 0 is its own predecessor), define the subtraction function.
7. 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.
8. Implement the function $n \mapsto \phi_n$ in λ-calculus.
9. Implement the predecessor function in OCaml and in λ-calculus.

IV Fixpoints

A *fixpoint combinator* is a term Y such that

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

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

11. 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.

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

13. Program a fixpoint combinator `fix` in OCaml.

14. Program the factorial function in OCaml.

15. Modify your implementation in order not to use the `rec` keyword, but you can use `fix`.

16. In practice, what happens when you evaluate the previous definition? Fix `fix`.

17. Define the factorial function in λ-calculus.

18. 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?