Normalizing in the λ-calculus

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

November 23, 2020

1 Termination of the simply typed λ-calculus

We recall the rules of the simply-typed λ-calculus:

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \Rightarrow B} \qquad \frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

where, in the first rule, we suppose $x \notin \mathrm{dom}(\Gamma')$. We want to show that every typable term t (in an arbitrary context) is *strongly normalizable*, meaning that there is no infinite reduction from t.

1. Can we show the property by induction on the derivation of the typing of t?

In the course of the proof, will need the following *well-founded induction* principle.

2. Suppose given a set X equipped with a binary relation $\to$ which is *well-founded*: there is no infinite sequence of reductions. Suppose given a property P on the elements of X such that, for every $t \in X$, we have

$$\forall t \in X.\ ((\forall t' \in X.\ t \to t' \Rightarrow P(t')) \Rightarrow P(t))$$

   Show that $\forall t \in X.\ P(t)$ holds. How can we recover recurrence as a particular case of this?

We define $\mathcal{R}(A)$, the *reducible* terms of type A, by induction by

- $\mathcal{R}(A)$, for A atomic, is the set of strongly normalizable terms,
- $\mathcal{R}(A \Rightarrow B)$ is the set of terms t such that $tu \in \mathcal{R}(B)$ for every $u \in \mathcal{R}(A)$.

A term is *neutral* when it is not an abstraction. We are going to show that following conditions hold:

(CR1) if $t \in \mathcal{R}(A)$ then t is strongly normalizable,
(CR2) if $t \in \mathcal{R}(A)$ and $t \to t'$ then $t' \in \mathcal{R}(A)$,
(CR3) if t is neutral and for every t' such that $t \to t'$ we have $t' \in \mathcal{R}(A)$ then $t \in \mathcal{R}(A)$.

3. Show that these conditions imply that a variable x belongs to $\mathcal{R}(A)$ for every type A.
4. Show the conditions (CR1), (CR2) and (CR3) by induction on A.
5. Suppose that $t[u/x] \in \mathcal{R}(B)$ for every $u \in \mathcal{R}(A)$. Show that $\lambda x.t \in \mathcal{R}(A \Rightarrow B)$.
6. Suppose that $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$ is derivable. Show that for all $u_1 \in \mathcal{R}(A_1)$, ..., $u_n \in \mathcal{R}(A_n)$, we have $t[u_1/x_1, \ldots, u_n/x_n] \in \mathcal{R}(A)$.
7. Show that all typable terms are reducible.
8. Show that all typable terms are strongly normalizable.
9. Use this to show that typable terms are confluent.

2 Normalization by evaluation

Implementing an evaluator for λ-calculus (or, more generally, for a functional programming language) is painful because one has to explicitly handle α-conversion. Techniques such as de Bruijn indices exist but they are quite error prone. We present here a technique called *normalization-by-evaluation* which allows easy implementation of normalization of λ-terms when the host language is itself functional and test for β-equivalence.

1. A term is *normal* when it cannot reduce. Give a grammar describing all terms in normal form.
2. A term is *neutral* when it is normal, and remains normal when applied to a normal form. Intuitively, this corresponds to a computation which is either finished or "stuck". Describe those by a grammar and use it to simplify the previous characterization of normal forms.
3. Define a function $[\![-]\!]_\rho$ which computes the normal form a term (we suppose that it is strongly normalizing) in an environment ρ which associates a normal form to free variables.
4. In OCaml define types corresponding to λ-terms, normal terms and neutral terms. If necessary, modify your implementation so that abstractions in neutral terms are implemented by OCaml abstractions. Finally, define a function `eval` which associates a normal term to every λ-term.
5. Suppose given a function `fresh` which generates fresh variable names. Implement a function `readback` which translates a normal form back to a λ-term.
6. Use this to implement a normalization function from λ-terms to λ-terms. Can we use it to easily test for β-conversion?
7. Transform your implementation in order to canonically generate variable names, so that the result is deterministic.
8. Extend the preceding constructions to products (and other constructors of your choice).

References

[1] Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 242–257, 1996.

[2] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*, volume 7. Cambridge university press Cambridge, 1989.