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?

   Solution. No, in the third rule we cannot show that if t and u are SN then tu also is, because a reduction in tu is not necessarily a reduction in t or a reduction in u (take $t = u = \lambda x.xx$).

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?

   Solution. By absurd, if there exists t_0 such that $\neg P(t_0)$, then there exists t_1 such that $t_0 \to t_1$ and $\neg P(t_1)$. Going on in this way, we construct an infinite sequence $t_0 \to t_1 \to t_2 \to \ldots$ such that $\neg P(t_i)$ for every index i, which is absurd.

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.

   Solution. The answer for this question and following ones can be found in [2, chapter 6].

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.

   Solution. Normal forms are generated by the grammar

   $$v ::= \lambda x.v \quad | \quad x\, v_1 \dots v_n$$

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.

   Solution. Neutral terms are of the form

   $$n ::= x \quad | \quad nv$$

   This can be defined mutually with the following definition for normal forms:

   $$v ::= \lambda x.v \quad | \quad n$$

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.

   Solution. We define

   $$\begin{aligned} [\![x]\!]_\rho &= \rho(x) \\ [\![\lambda x.t]\!]_\rho &= \lambda v.[\![t]\!]_{\rho[x\mapsto v]} \\ [\![t\, u]\!]_\rho &= [\![t]\!]_\rho\, [\![u]\!]_\rho \end{aligned}$$

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.

   Solution.

```
type var = string

type term =
  | Var of var
  | Abs of var * term
  | App of term * term

type value =
  | VAbs of (value -> value)
  | VNeu of neutral
and neutral =
  | NVar of var
  | NApp of neutral * value

let rec eval env = function
  | Var x -> (try List.assoc x env with Not_found -> VNeu (NVar x))
  | Abs (x, t) -> VAbs (fun v -> eval ((x,v)::env) t)
  | App (t, u) ->
    let u = eval env u in
    match eval env t with
    | VAbs f -> f u
    | VNeu n -> VNeu (NApp (n, u))
```

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.

   Solution.

```
let fresh =
  let n = ref 0 in
  fun () -> incr n; "x" ^ string_of_int !n

let rec readback = function
  | VAbs f ->
    let x = fresh () in
    Abs (x, readback (VNeu (NVar x)))
  | VNeu n -> readback_neutral n
and readback_neutral = function
  | NVar x -> Var x
  | NApp (n, v) -> App (readback_neutral n, readback v)
```

6. Use this to implement a normalization function from λ-terms to λ-terms. Can we use it to easily test for β-conversion?

   Solution. We normalize with

```
let normalize t = readback 0 (eval [] t)
```

   In order to test for β-conversion we would still need to implement α-conversion.

7. Transform your implementation in order to canonically generate variable names, so that the result is deterministic.

   Solution. We keep an integer which we use to generate variable names and is incremented on abstractions only. Implement a tests for β-convertibility for strongly normalizing terms.

```
let fresh i = "x" ^ string_of_int i

let rec readback i = function
  | VAbs f ->
    let x = fresh i in
    Abs (x, readback (i+1) (VNeu (NVar x)))
  | VNeu n -> readback_neutral i n
and readback_neutral i = function
  | NVar x -> Var x
  | NApp (n, v) -> App (readback_neutral i n, readback i v)
```

   We can then test for β-convertibility by comparing normal forms:

```
let eq t u = normalize t = normalize u
```

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.