Lambda Calculus Evaluator in OCaml

Christophe Deleuze

11 April, 2008, revised Jan 2022

1. Introduction

This document is a lambda calculus evaluator with tracing facilities, coded in Objective Caml [3]. It is written as a literate program, using the ocamlweb [2] tool.

2. Lambda terms

Here is the type of lambda terms. We use the "classical" form with variable names, so we will have to perform alpha conversion sometimes. An alternative would be to use De Bruijn's nameless dummies [1]. *Hole* and *Here* are not part of lambda terms proper but will allow to define contexts (used when tracing reductions).

```
type term =
    Var of string | Abs of string × term | App of term × term
  | Hole | Here of term
```

3. Alpha-conversion

To perform alpha conversion, we'll need to know what are the free variables in a term t. These are variables appearing in t except for occurences appearing under a lambda binding them. We simply concat lists of variables appearing in subterms, filtering out the bound variable at each *Abs* node. We don't mind that variables may appear multiple times in our list.

```
let rec fv t =
  match t with
  | Var x → [ x ]
  | Abs(x, l) → List.filter (fun e → e ≠ x) (fv l)
  | App(t1, t2) → fv t1 @ fv t2
```

So here is alpha conversion. In term t this function renames all bound variables whose name is in *names*. Renaming is performed by suffixing *nb* to the variable name. A fresh suffix will have to be provided at each call.

```
let alpha names nb t =
  let rec alpha bound t =
    match t with
      | Var s → if List.mem s bound then Var (s^nb) else t
      | App(l1, l2) → App(alpha bound l1, alpha bound l2)
      | Abs(s, l) →
        if List.mem s names then
          Abs(s^nb, alpha (s :: bound) l)
```

```
        else
          Abs(s, alpha bound l)
  in
  alpha [] t
```

4. Beta-reduction
Here is substitution, performing body[s/arg] but assuming no variable captures can occur, i.e. alpha conversion has already been performed if necessary.

```
let rec ssubst body s arg =
  match body with
  | Var s' → if s = s' then arg else Var s'
  | App(l1, l2) → App(ssubst l1 s arg, ssubst l2 s arg)
  | Abs(o, l) → if o = s then body else Abs(o, ssubst l s arg)
```

Now, the real beta reduction, first performing alpha conversion to avoid variable captures. *gen_nb* provides the unique number for the suffix we mentionned above; *init_nb* resets the generator. What variables can be captured? These are free variables in *arg* that are bound in *body*. So we rename in *body* bound variables that appear free in *arg*.

```
let gen_nb, init_nb =
  let nb = ref 0 in (fun () → incr nb; !nb), (fun () → nb := 0)

let beta (App(Abs(s, body), arg)) =
  let nb = "~" ^ string_of_int (gen_nb()) in
  ssubst (alpha (fv arg) nb body) s arg
```

5. Evaluation
We now have what we need to build evaluation functions. Here's call by name.

```
let rec cbn t =
  match t with
  | Var _ → t
  | Abs _ → t
  | App(e1, e2) → let e1' = cbn e1 in
    match e1' with
    | Abs _ → cbn (beta (App(e1', e2)))
    | _ → App(e1', e2)
```

Normal order evaluation can be built on top of call by name, as is shown in [4].

```
let rec nor t =
  match t with
  | Var _ → t
  | Abs(x, e) → Abs(x, nor e)
  | App(e1, e2) → let e1' = cbn e1 in
    match e1' with
    | Abs _ → nor (beta (App(e1', e2)))
    | _ → let e1'' = nor e1' in
      App(e1'', nor e2)
```

6. Printing lambda terms

Lambda symbols will be printed as slashes '/'. Printing of terms can be performed by a simple recursive traversal. However, since such printing is intended for humans to read, we want to print them somehow nicely, taking into account the following two rules:

- do not print parentheses when not necessary: application is left-associative,
- successive abstractions are grouped: print `/fx.fx` instead of `/f./x.fx`

For this we need a little bit of context information to decide how to precisely print a subterm, here it is:

- *Root*: current term has no father,
- *Lambda*: current term is just under a lambda,
- *LApp*: current term is the left son of an *App* node,
- *RApp*: current term is the right son of an *App* node.

As we previously said *Hole* and *Here* are for contexts, we will come back to this later.

```
type upnode = Root | Lambda | LApp | RApp

let rec string_of_term t =
  let rec sot up = function
    | Var s → s
    | Abs(v, body) → begin
        let p, s = match up with
        | Lambda → "", ""
        | Root → "/",""
        | LApp
        | RApp → "(/",")"
        in
        p ^ v ^ (match body with Abs _ → "" | _ → ".") ^ sot Lambda body ^ s
     end
    | App(f, arg) →
        let p, s = if up = RApp then "(",")" else "","" in
        p ^ (sot LApp f) ^ (sot RApp arg) ^ s

    | Hole → " <> "
    | Here l → " < " ^ string_of_term l ^ " > "
  in
  sot Root t
```

7. Environment

For convenience we want to be able to refer to a set of named terms, which we'll call the environment. We implement it as a global hash table binding strings to *terms*.

```
let env = Hashtbl.create 30
```

```
let get_env n =
  try
    Hashtbl.find env n
  with _ → failwith ("not in env: " ^ n)

let set_env n t =
  Hashtbl.add env n t
```

8. Parsing lambda terms: lexical
For parsing, printing, and user convenience, names refering to the environment are introduced by the underscore character like in `/x._fact x` and end-delimited by any non letter character. We prefer this to using spaces so that we can still use sequences of (single characters) variable names without interleaved spaces (as in `/fx.f(fx)`).
We start with the lexical analyzer. It'll get a string and produce a stream of lexical *tokens*.

```
type token = CHAR of char | LPAR | RPAR | DOT | LAMBDA | STRING of string
           | END
```

For example, from `/x._fact x` we'll get *LAMDBA*, *CHAR* `'x'`, *DOT*, *STRING* `"fact"`, *CHAR* `'x'`.

soc is short for *string_of_char*. *implode* turns a list of characters into a string made of these characters in the reverse order. This will be used to build *STRING* tokens.

```
let soc = Printf.sprintf "%c"

let implode l =
  let rec imp l acc = match l with
    | [] → acc
    | h :: t → imp t ((soc h)^acc) in imp l ""
```

get_name reads characters from the character stream and cons-accumulates them until a non letter character is found. It returns the string of the accumulated characters in the initial order by calling *implode*.

```
let get_name s =
  let rec loop acc =
    let n = Stream.peek s in
    if n = None then acc
    else let c = match n with Some c → c in
          match c with
          | '_' | '/' | '(' | ')' | '.' | ' ' → acc
          | _ → Stream.next s;
                loop (c :: acc)
  in
  implode (loop [])
```

The lexer function takes a string and returns a token stream. It first turns the string into a character stream, then consumes them producing tokens. When an underscore character is found, all following letters are accumulated by *get_name* to form a *STRING* token. Blanks are simply skipped. All other characters directly map to a token. The *END* token is produced when string end is reached.

```
let lexer s = let s = Stream.of_string s in let rec next _ = let n =
    Stream.peek s in if n = None then Some END else match Stream.next s
    with | '_' → Some(STRING (get_name s)) | '/' → Some LAMBDA | '('
    → Some LPAR | ')' → Some RPAR | '.' → Some DOT | ' ' → next 0 |
    c → Some (CHAR c) in Stream.from next
```

9. Parsing lambda terms: syntax
We now turn to the parsing proper. Our LL(1) grammar for lambda terms is:

full	$\rightarrow$	term *END*
term	$\rightarrow$	elt elts | lamb
elts	$\rightarrow$	elt elts | ε
elt	$\rightarrow$	(*CHAR c*) | *LPAR* term *RPAR* | (*STRING s*)
lamb	$\rightarrow$	*LAMBDA* (*CHAR c*) lamb2
lamb2	$\rightarrow$	*DOT* term | (*CHAR c*) lamb2

full is to ensure we parse the term from the full entry and not just a prefix thereof. Each non atomic element (ie not a single char variable or underscore-started name) must be enclosed in parenthesis, except for a top-level lambda. We want application to be left associative. The lamb2 rule allows compact notation for a sequence of lambdas, so that eg `/fnx.x` will be parsed this way:

term → lamb → `/f` lamb2 → `/fn` lamb2 → `/fnx` lamb2
→ `/fnx.`term → `/fnx.`elt elts → `/fnx.x` elts → `/fnx.x` ε

We build our top-down parser using Caml streams, so we'll need camlp4 to use stream syntax. This is taken care of in the Makefile; in an interactive session we could use:

```
#use "topfind";;
#camlp4o;;
```

Caml built-in parsers exactly mimic the grammar. Top-down parsers "naturally" making operators right-associative (by not allowing left recursion in the grammar rules), note how *App* is parsed as left-associative by using an accumulator in *elts* (tree for successive applications is built left-to-right, bottom-up)[1]

```
let rec full = parser
 | [⟨ t = term; 'END ⟩] → t
and term = parser
 | [⟨ e = elt; t = elts e ⟩] → t
 | [⟨ l = lamb ⟩] → l
and elts e1 = parser
 | [⟨ e2 = elt; e = elts (App(e1, e2)) ⟩] → e
 | [⟨ ⟩] → e1
and elt = parser
 | [⟨ 'CHAR c ⟩] → Var (soc c)
 | [⟨ 'LPAR; t = term; 'RPAR ⟩] → t
```

[1] Of course this is only possible because it's a degenerated tree.

```
  | [⟨ 'STRING s ⟩] → get_env s
and lamb = parser
  | [⟨ 'LAMBDA; 'CHAR c; s = lamb2 ⟩] → Abs(soc c, s)
and lamb2 = parser
  | [⟨ 'DOT; t = term ⟩] → t
  | [⟨ 'CHAR c; s = lamb2 ⟩] → Abs(soc c, s)
```

Here is the final function for turning a string into a term.

```
let term_of_string s = full (lexer s)
```

10. Tracing reductions
In order to trace reductions, we will need the following:

- the beta reduction function will have to print the whole term before and after reduction,
- the printed term should show somehow the current redex,
- this means the beta function should receive the redex to reduce *along with its context* in order to print the whole term; and the evaluations function (*cbn* or *nor*) will have to maintain the context of the current recursively explored term.

A context is a lambda term with a single *Hole* (a placeholder for the term whose context it is). A subterm in context is a term appearing under a *Here* node in its context term. See code of *string_of_term* above to see the string representation.
put_in_hole puts expression e in hole of context c. If e is a term proper, the result is a term proper. If e is a context, the result is a new (extended) context.

```
let put_in_hole c e =
  let rec pih c =
    match c with
    | Hole → e
    | Abs(s, Hole) → Abs(s, e)
    | App(e1, Hole) → App(e1, e)
    | App(Hole, e2) → App(e, e2)
    | Abs(s, o) → Abs(s, pih o)
    | App(o1, o2) → App(pih o1, pih o2)
    | Var _ → c
  in
  pih c
```

We have to maintain the context during recursive evaluation, but using *put_in_hole* at each step would be very costly. Instead, we will accumulate a list of context steps, and build the corresponding context only when we need it.

buildc builds the context from a list of context steps. This is done by putting the last step in the hole of previous one, putting the obtained term in hole of previous context step etc.

```
let buildc c =
  let rec soc acc c =
    match c with
    | [] → acc
    | h :: t → soc (put_in_hole acc h) t
  in
  soc Hole (List.rev c)
```

Now, given a list of context steps c and an expression e, we build the term showing e in its context. That is, we insert e under a *Here* in the context built from the list of context steps.

```
let put_in_context c e =
  match c with
  | h :: t → buildc ((put_in_hole h (Here e)) :: t)
  | _ → Here e
```

This one just puts e at its place in c, without adding a *Here* node.

```
let plug_in_context c e =
  match c with
  | h :: t → buildc ((put_in_hole h e) :: t)
  | _ → e
```

Evaluation functions will take as argument a function *beta* that will perform the beta reduction on the given sub-term. It will receive as first argument a list of context steps that it can use as a possible side-effect.
Here, *tsub* prints the term in context before reduction, performs reduction, prints the reduced term in context, prints the whole reduced term with context marks and returns the reduced term. *tsubf* prints only the resulting reduced term.

```
let tsub ctxt t =
  print_string ("> " ^ (string_of_term (put_in_context ctxt t)) ^ "\n");
  let t' = beta t in
  print_string ("< " ^ (string_of_term (put_in_context ctxt t')) ^ "\n");
  print_string ("= " ^ (string_of_term (plug_in_context ctxt t')) ^ "\n");
  t'
```

```
let tsubf ctxt t =
  let t' = beta t in
  print_string ("= " ^ (string_of_term (plug_in_context ctxt t')) ^ "\n");
  t'
```

tsub2 does the same thing as *tsub* but waits for the return key to be pressed between each step.

```
let key () = flush stdout; input_char stdin
```

```
let tsub2 ctxt t =
  print_string ("> " ^ (string_of_term (put_in_context ctxt t)) ^ "\n");
  key();
  let t' = beta t in
  print_string ("< " ^ (string_of_term (put_in_context ctxt t')) ^ "\n");
  key();
  print_string ("= " ^ (string_of_term (plug_in_context ctxt t')) ^ "\n");
  key();
  t'
```

11. New evaluation functions

We rewrite our evaluation functions so that they maintain context steps, take the beta reduction function as a parameter and provide it the context as well as the redex. Here's call by name (we need to pass a context steps arg *ctxt* because *cbn* can be called from *nor* below):

```
let cbn beta ctxt t =
  let rec cbn ctxt t =
    match t with
    | Var _ → t
    | Abs _ → t
    | App(e1, e2) →
       let e1' = cbn (App(Hole, e2) :: ctxt) e1 in
       match e1' with
         Abs _ → cbn ctxt (beta ctxt (App(e1', e2)))
       | _ → App(e1', e2)
  in
  cbn ctxt t
```

And normal order evaluation:

```
let nor beta t =
  let rec nor ctxt t =
    match t with
    | Var _ → t
    | Abs(x, e) → Abs(x, nor (Abs(x, Hole) :: ctxt) e)
    | App(e1, e2) →
       let e1' = cbn beta (App(Hole, e2) :: ctxt) e1 in
       match e1' with
         Abs _ → nor ctxt (beta ctxt (App(e1', e2)))
       | _ → let e1'' = nor (App(Hole, e2) :: ctxt) e1'
             in App(e1'', nor (App(e1'', Hole) :: ctxt) e2)
  in
  nor [] t
```

Traced and stepped normal order evaluation. We reset the number generator at each use:

```
let trace s = init_nb(); nor tsubf (term_of_string s)
```

```
let step s = init_nb(); nor tsub2 (term_of_string s)
```

Non traced normal order evaluation. This one does not use the context steps that are being accumulated.

```
let nnor s = init_nb(); nor (fun c t → beta t) (term_of_string s)
```

12. Basic constructs
To be able to play with the system, we define some useful basic constructs.

```
let add_env n s = set_env n (term_of_string s)

add_env "succ" "/nfx.f(nfx)";
add_env "pred" "/nfx.n(/gh.h(gf))(/u.x)(/u.u)";
add_env "mult" "/nm./fx.n(mf)x";
add_env "exp" "/mn.nm";
add_env "zero" "/fx.x";
add_env "true" "/xy.x";
add_env "false" "/xy.y";
add_env "iszero" "/n.n(/x._false)(_true)";
add_env "Y" "/g.(/x.g(xx))(/x.g(xx))";
```

We now have all we need to define the factorial function.

```
add_env "ofact" "/fn.(_iszero n)(_succ _zero)(_mult n(f(_pred n)))";
add_env "fact" "_ofact(_Y _ofact)"
```

13. Using it
We're mostly done. Let's create a simple shell to play with lambda terms. Here's first the definition of a few commands to alter the environment and select the evaluation function:

```
let cont = ref true
let eval = ref trace

let command (cmd :: args) =
  match cmd, args with
    ":env", [] → Hashtbl.iter (fun k v → Printf.printf "%s = %s\n" k (string_of_term v)) env
  | ":quit",[] → cont := false
  | ":add",[n; v] → add_env n v
  | ":trace",[] → eval := trace
  | ":step",[] → eval := step
  | ":nnor",[] → eval := nnor
  | ":show",l → print_endline (string_of_term (term_of_string (String.concat " " l)))
  | _ → print_endline "unknown command or bad args"
```

And finally a simple interactive loop that reads a line, executes it if it's a command and otherwise evaluates it as a lambda term and prints the result.

```
let rec loop () =
  print_string "/> ";
  try
    let l = read_line () in
    if l ="" then () else
      begin
        if l.[0] = ':' then command (String.split_on_char ' ' l) else
        if l.[0] = ';' then loop() else
          let r = string_of_term (!eval l) in
```

```
            print_endline r
        end;
      if !cont then loop()
  with _ → print_endline "Syntax error"; loop()
loop()
```

Let's try a few examples (default evaluation function is *trace*):

```
/> _succ /nfx.f(nfx)
Syntax error
/> _succ (/fx.f(fx))
= /fx.f((/fx.f(fx))fx)
= /fx.f((/x.f(fx))x)
= /fx.f(f(fx))
/fx.f(f(fx))
```

Let's multiply two by three...

```
/> :add two /fx.f(fx)
/> :show _mult _two (_succ _two)
(/nmfx.n(mf)x)(/fx.f(fx))((/nfx.f(nfx))(/fx.f(fx)))
/> ; all right, compute that!
/> _mult _two (_succ _two)
= (/mfx.(/fx.f(fx))(mf)x)((/nfx.f(nfx))(/fx.f(fx)))
= /fx.(/fx.f(fx))((/nfx.f(nfx))(/fx.f(fx))f)x
= /fx.(/x.(/nfx.f(nfx))(/fx.f(fx))f((/nfx.f(nfx))(/fx.f(fx))fx))x
= /fx.(/nfx~4.f(nfx~4))(/fx~4.f(fx~4))f((/nfx~4.f(nfx~4))(/fx~4.f(fx~4))fx)
= /fx.(/fx~4.f((/fx~4.f(fx~4))fx~4))f((/nfx~4.f(nfx~4))(/fx~4.f(fx~4))fx)
= /fx.(/x~4.f((/f~6x~4.f~6(f~6x~4))fx~4))((/nfx~4.f(nfx~4))(/fx~4.f(fx~4))fx)
= /fx.f((/f~6x~4.f~6(f~6x~4))f((/nfx~4.f(nfx~4))(/fx~4.f(fx~4))fx))
= /fx.f((/x~4.f(fx~4))((/nfx~4.f(nfx~4))(/fx~4.f(fx~4))fx))
= /fx.f(f(f((/nfx~4.f(nfx~4))(/fx~4.f(fx~4))fx)))
= /fx.f(f(f((/fx~4.f((/fx~4.f(fx~4))fx~4))fx)))
= /fx.f(f(f((/x~4.f((/f~11x~4.f~11(f~11x~4))fx~4))x)))
= /fx.f(f(f(f((/f~11x~4.f~11(f~11x~4))fx))))
= /fx.f(f(f(f((/x~4.f(fx~4))x))))
= /fx.f(f(f(f(f(fx)))))
/fx.f(f(f(f(f(fx)))))
```

... which is six! What about the factorial function?

```
/> _fact (/fx.fx)
= (/n.(/n.n(/xxy.y)(/xy.x))n((...
...
= /fx.(/x~12.f((/f~30x~12.x~12)fx~12))x
= /fx.f((/f~30x~12.x~12)fx)
= /fx.f((/x~12.x~12)x)
= /fx.fx
```

```
/fx.fx

/> _fact (/fx.f(f(fx)))
= (/n.(/n.n(/xxy.y)(/xy.x))n((/nfx.f(nfx))(/fx.x))((...
... 673 steps skipped
= /fx.f(f(f(f(f(fx)))))
/fx.f(f(f(f(f(fx)))))
```

Good, fact 3 is indeed 6!

References

[1] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392, 1972. `http://www.win.tue.nl/automath/archive/pdf/aut029.pdf`

[2] Jean-Christophe Filliâtre and Claude Marché. ocamlweb: a literate programming tool for objective caml. `https://www.lri.fr/~filliatr/ocamlweb/`

[3] INRIA. OCaml. `https://ocaml.org`

[4] Peter Sestoft. Demonstrating lambda calculus reduction. In *The essence of computation: complexity, analysis, transformation*, pages 420–435. Springer-Verlag New York, Inc., New York, NY, USA, 2002.