Electronic Notes in Theoretical Computer Science 45 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume45.html` 9 pages

Demonstrating Lambda Calculus Reduction

Peter Sestoft [1]

Department of Mathematics and Physics
Royal Veterinary and Agricultural University, Denmark

and

IT University of Copenhagen, Denmark

Abstract

We describe lambda calculus reduction strategies using big-step operational semantics and show how to efficiently trace such reductions. This is used in a web-based lambda calculus reducer, at `http://www.dina.kvl.dk/~sestoft/lamreduce/`.

1 Introduction

The pure untyped lambda calculus [2] is often taught as part of the computer science curriculum. It may be taught in a computability course as a classical computation model [3]. It may be taught in a semantics course as the foundation for denotational semantics. It may be taught in a functional programming course as the archetypical minimal functional programming language. It may be taught in a programming language concepts course for the same reason, or to demonstrate that a very small language can be universal, e.g. can encode arithmetics (as well as data structures, recursive function definitions and so on), using encodings such as these:

$$\begin{aligned} two &\equiv \lambda f.\lambda x.f(fx) \\ four &\equiv \lambda f.\lambda x.f(f(f(fx))) \\ add &\equiv \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx) \end{aligned} \tag{1}$$

This paper is motivated by the assumption that to appreciate the operational aspects of pure untyped lambda calculus, students must experiment with it, and that tools encourage experimentation with encodings and reduction strategies by making it less tedious and more fun.

In this paper we describe a simple way to create a tool for demonstrating lambda calculus reduction. Instead of describing a reduction strategy by a

[1] `sestoft@dina.kvl.dk`, Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark.

procedure for locating the next redex to be contracted, we describe it by a big-step operational semantics. We show how to trace the β-reductions performed during reduction.

We also discuss the relation between programming language concepts such as call-by-name and call-by-value, and lambda calculus concepts such as normal order reduction and applicative order reduction.

2 The Pure Untyped Lambda Calculus

We use the pure untyped lambda calculus [2]. A lambda term is a variable x, a lambda abstraction $\lambda x.e$ which binds x in e, or an application $(e\,e)$:

$$e ::= \; x \mid \lambda x.e \mid e\,e \tag{2}$$

Lambda terms may have free variables, not bound by any enclosing lambda abstraction. Term identity $e_1 \equiv e_2$ is taken modulo renaming of lambda-bound variables, as usual. The notation $e[e_x/x]$ denotes substitution of e_x for x in e, with renaming of bound variables to avoid capture.

3 Functional Programming Languages

In practical functional programming languages such as Scheme [11], Standard ML [6] or Haskell [10], programs (terms) cannot have free variables, and reductions are not performed under lambda abstractions or other variable binders, as this would considerably complicate their efficient implementation [9].

However, an implementation of lambda calculus reduction must perform reductions under lambda abstractions. Otherwise, *add two two* would not reduce to *four* using the encodings (1), which would disappoint students.

Because free variables and reduction under abstraction are absent in functional languages, it is unclear what the programming language concepts call-by-value and call-by-name mean in the lambda calculus. In particular, how should free variables be handled, and to what normal form should call-by-value and call-by-name evaluate? We propose the following answers:

- A free variable is similar to a data constructor (in Standard ML or Haskell), that is, an uninterpreted function symbol. If the free variable is in function position $(x\,e_2)$, then call-by-value should reduce the argument expression e_2, whereas call-by-name should not. This is consistent with constructors being strict in strict languages (e.g. ML) and non-strict in non-strict languages (e.g. Haskell).
- Functional languages perform no reduction under abstractions, and thus reduce to weak normal forms only. In particular, call-by-value reduces to weak normal form, and call-by-name reduces to weak head normal form.

4 Lazy Functional Programming Languages

Under lazy evaluation, a variable-bound term is evaluated at most once, regardless how often the variable is used [9]. This evaluation mechanism may be called call-by-need, or call-by-name with sharing of argument evaluation. Lazy languages also permit the creation of cyclic terms, or cycles in the heap, by definitions such as this, which creates a finite (cyclic) representation of an infinite list of 1's:

```
val ones = 1 :: ones
```

Substitution of terms for variables cannot truly model such constant-size cyclic structures created by lazy evaluation, only approximate it by unbounded unfolding of a recursive term definition (e.g. encoded using some version of the recursion combinator Y). To properly express sharing of subterm evaluation, and the creation of cyclic terms, one must extend the calculus (2) with mutually recursive bindings:

(3) $$e ::= \ x \mid \lambda x.e \mid e\, e \mid \mathit{letrec}\ \{x_i = e_i\}\ \mathit{in}\ e$$

The sharing of subterm evaluation and the creation of cyclic terms may be modelled using either graph reduction (Wadsworth 1971, Turner 1979, and subsequent work [1,9]), or an explicit heap [5,12].

In any case, proper modelling of lazy evaluation requires syntactic extensions as well as a more complicated evaluation model than just term reduction. We shall not consider lazy evaluation any further in this paper, and shall consider only the syntax in (2) above.

5 Normal Forms

We need to distinguish four different normal forms, depending on whether we reduce under abstractions (in the lambda calculus) or not (in functional programming languages), and depending on whether we reduce the arguments before substitution (in strict languages) or not (in non-strict languages).

The table below summarizes the four normal forms using context-free grammars. The grammar symbol E denotes a term in the relevant normal form, e denotes an arbitrary lambda term generated by (2), and $n \geq 0$. Note how the two dichotomies generate the four grammars just by varying e or E:

	Reduce under abstractions	
Reduce arguments	Yes	No
Yes	Normal form $E ::= \ \lambda x.E \mid x\, E_1 \ldots E_n$	Weak normal form $E ::= \ \lambda x.e \mid x\, E_1 \ldots E_n$
No	Head normal form $E ::= \ \lambda x.E \mid x\, e_1 \ldots e_n$	Weak head normal form $E ::= \ \lambda x.e \mid x\, e_1 \ldots e_n$

6 Reduction Strategies and Reduction Functions

We present some reduction strategies using big-step operational semantics, or natural semantics [4], and their implementation in Standard ML. We exploit that Standard ML has a well-defined semantics [6]: it evaluates a function's arguments before calling the function, it evaluates the right-hand side of `let`-bindings before binding the variable, and it evaluates terms from left to right.

We model lambda terms x, $\lambda x.e$ and $(e\,e)$ as ML constructed data, representing variables by strings:

```
datatype lam = Var of string
             | Lam of string * lam
             | App of lam * lam
```

We also assume an auxiliary function `subst : lam -> lam -> lam` that implements capture-free substitution, so `subst ex (Lam(x, e))` is the ML representation of $e[e_x/x]$, the result of β-reduction of $(\lambda x.e)\,e_x$.

6.1 Call-by-name Reduction

Call-by-name reduction $e \xrightarrow{bn} e'$ is leftmost weak reduction:

$$x \xrightarrow{bn} x$$

$$(\lambda x.e) \xrightarrow{bn} (\lambda x.e)$$

$$\frac{e_1 \xrightarrow{bn} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{bn} e'}{(e_1\ e_2) \xrightarrow{bn} e'} \tag{4}$$

$$\frac{e_1 \xrightarrow{bn} e_1' \not\equiv \lambda x.e}{(e_1\ e_2) \xrightarrow{bn} (e_1'\ e_2)}$$

It is easily seen that all four rules generate terms in weak head normal form. The following ML function `cbn` computes the weak head normal form of a lambda term, contracting redexes in the order implicit in the operational semantics (4) above. The two first function clauses below implement the two first semantics rules above. The third function clause below implements the third and fourth rule, discriminating on the result of reducing e_1:

```
fun cbn (Var x)        = Var x
  | cbn (Lam(x, e))    = Lam(x, e)
  | cbn (App(e1, e2)) =
    case cbn e1 of
        Lam (x, e) => cbn (subst e2 (Lam(x, e)))
      | e1'        => App(e1', e2)
```

6.2 Normal Order Reduction

Normal order reduction $e \xrightarrow{no} e'$ is leftmost reduction. The function term e_1 in an application $(e_1\, e_2)$ must be reduced using call-by-name (4). Namely, if e_1 reduces to an abstraction $(\lambda x.e)$, then $((\lambda x.e)\, e_2)$ is a redex outside any redex in e, and must be reduced first.

(5)

$$x \xrightarrow{no} x$$

$$\frac{e \xrightarrow{no} e'}{(\lambda x.e) \xrightarrow{no} (\lambda x.e')}$$

$$\frac{e_1 \xrightarrow{bn} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{no} e'}{(e_1\ e_2) \xrightarrow{no} e'}$$

$$\frac{e_1 \xrightarrow{bn} e_1' \not\equiv (\lambda x.e) \qquad e_1' \xrightarrow{no} e_1'' \qquad e_2 \xrightarrow{no} e_2'}{(e_1\ e_2) \xrightarrow{no} (e_1''\ e_2')}$$

These rules are easily seen to generate normal form terms only. The implementation of the reduction strategy as a function `nor` in Standard ML is straightforward. It uses the function `cbn` from Section 6.1 above:

```
fun nor (Var x)        = Var x
  | nor (Lam (x, e))   = Lam(x, nor e)
  | nor (App(e1, e2)) =
    case cbn e1 of
        Lam(x, e) => nor (subst e2 (Lam(x, e)))
      | e1'       => let val e1'' = nor e1'
                     in App(e1'', nor e2) end
```

6.3 Call-by-value Reduction

Call-by-value reduction $e \xrightarrow{bv} e'$ is defined below. It differs from call-by-name (Section 6.1) only by reducing the argument of an application $(e_1\, e_2)$ before contracting the redex, and before building an application term:

(6)

$$\frac{e_1 \xrightarrow{bv} (\lambda x.e) \qquad e_2 \xrightarrow{bv} e_2' \qquad e[e_2'/x] \xrightarrow{bv} e'}{(e_1\ e_2) \xrightarrow{bv} e'}$$

$$\frac{e_1 \xrightarrow{bv} e_1' \not\equiv (\lambda x.e) \qquad e_2 \xrightarrow{bv} e_2'}{(e_1\ e_2) \xrightarrow{bv} (e_1'\ e_2')}$$

These rules are easily seen to generate weak normal form terms only. The implementation of the rules by an ML function is straightforward and is omitted.

6.4 Applicative Order Reduction

Applicative order reduction $e \xrightarrow{ao} e'$ is defined below. It differs from call-by-value (Section 6.3) only by reducing under abstractions. The rules are easily seen to generate only normal forms:

$$\frac{e \xrightarrow{ao} e'}{(\lambda x.e) \xrightarrow{ao} (\lambda x.e')} \tag{7}$$

7 Tracing: Side-Effecting Substitution, and Contexts

The reducers defined in ML above perform the substitutions $e[e_2/x]$ in the same order as prescribed by the operational semantics, thanks to Standard ML semantics: strict evaluation and left-to-right evaluation. But they only return the final reduced lambda term; they do not trace the intermediate steps of the reduction, which is often more interesting from a pedagogical point of view.

ML permits expressions to have side effects, so we can make the substitution function report (e.g. print) the redex just before contracting it. To do this we define a modified substitution function `csubst` which takes as argument another function `c` and applies it to the redex `App(Lam(x, e), ex)` representing $(\lambda x.e)\, e_x$ just before contracting it:

```
fun csubst (c : lam -> unit) ex (Lam(x, e)) =
    (c (App(Lam(x, e), ex));
     subst ex (Lam(x, e)))
```

The function `c : lam -> unit` is evaluated for its side effect only, as shown by the trivial result type `unit`. Evaluating `csubst c ex (Lam(x, e))` has the *effect* of calling `c` on the redex `App(Lam(x, e), ex)`, and the *result* of evaluating `subst ex (Lam(x, e))`, which is the contracted redex.

We can define a function `printlam : lam -> unit` that prints the lambda term as a side effect. Then we can replace the call `subst e2 (Lam(x, e))` in `cbn` of Section 6.1 by `csubst printlam e2 (Lam(x, e))`. Then the reduction of a term by `cbn` will produce a printed trace of all the redexes $((\lambda x.e)\, e_x)$ in the order in which they are contracted.

This still does not give us a usable trace of the evaluation: we do not know where in the current term the redex in question occurs. This is because the function `c` is applied only to the redex itself; the term surrounding the redex is implicit. To make the context of the redex explicit, we can use *contexts*, or terms with a single hole, such as $\lambda x.[\,]$ and $(e_1\,[\,])$ and $([\,]\,e_2)$. Filling the hole of a context with a lambda term produces a lambda term. The following grammar generates all contexts:

$$C \;::=\; [\,] \mid \lambda x.C \mid e\,C \mid C\,e \tag{8}$$

A context can be represented by an ML function of type `lam -> lam`. The four forms of contexts (8) can be built using four ML context-building functions:

```
fun id      e  = e
fun Lamx x  e  = Lam(x, e)
fun App2 e1 e2 = App(e1, e2)
fun App1 e2 e1 = App(e1, e2)
```

For instance, `App1` e_2 is the ML function `fn e1 => App(e1, e2)` which represents the context $([\,]\, e_2)$. Filling the hole with the term e_1 is done by computing `(App1` e_2`)` e_1 which evaluates to `App(`e_1`,` e_2`)`, representing $(e_1\, e_2)$.

Function composition (`f o g`) composes contexts. For instance, the composition of contexts $\lambda x.[\,]$ and $([\,]\, e_2)$ is `Lamx` x `o App1` e_2, which represents the context $\lambda x.([\,]\, e_2)$. Similarly, the composition of the contexts $([\,]\, e_2)$ and $\lambda x.[\,]$ is `App1` e_2 `o Lamx` x, which represents $((\lambda x.[\,])\, e_2)$.

8 Reduction in Context: Call-by-name

To produce a trace of the reduction, we modify the reduction functions defined in Section 6 to take an extra context argument `c` and to use the extended substitution function `csubst`, passing `c` to `csubst`. Then `csubst` will apply `c` to the redex before contracting it. We take the call-by-name reduction function `cbn` (Section 6.1) as an example; the other reduction functions are handled similarly. The reduction function must build up the context `c` as it descends into the term. It does so by composing the context with the appropriate context builder (in this case, only in the `App` branch):

```
fun cbnc c (Var x)        = Var x
  | cbnc c (Lam(x, e))    = Lam(x, e)
  | cbnc c (App(e1, e2)) =
    case cbnc (c o App1 e2) e1 of
        Lam (x, e) => cbnc c (csubst c e2 (Lam(x, e)))
      | e1'        => App(e1', e2)
```

By construction, if c `: lam -> lam` and the evaluation of `cbnc` $c\ e$ involves a call `cbnc` $c'\ e'$, then $c[e] \longrightarrow_\beta^* c'[e']$. Also, whenever a call `cbnc` $c'\ (e_1\, e_2)$ is evaluated, and $e_1 \xrightarrow{bn} (\lambda x.e)$, then function c' is applied to the redex $((\lambda x.e)\, e_2)$ just before it is contracted. Hence a trace of the reduction of term `e` can be obtained just by calling `cbnc` as follows:

```
cbnc printlam e
```

where `printlam : lam -> unit` is a function that prints the lambda term as a side effect. In fact, computing `cbnc printlam (App (App` *add two*`)` *two*`)`, using the encodings from (1), prints the two intermediate terms below. The third term shown is the final result (a weak head normal form):

```
(\m.\n.\f.\x.m f (n f x)) (\f.\x.f (f x)) (\f.\x.f (f x))
(\n.\f.\x.(\f.\x.f (f x)) f (n f x)) (\f.\x.f (f x))
\f.\x.(\f.\x.f (f x)) f ((\f.\x.f (f x)) f x)
```

A web-based interface can be created by defining a function `htmllam` that prints HTML code, and calling `cbnc` from a CGI script on the webserver with `htmllam` as argument. Such an implementation written in Moscow ML [7] is available at `http://www.dina.kvl.dk/~sestoft/lamreduce/`.

9 Single-stepping Reduction

For experimentation it is useful to be able to perform one beta-reduction at a time, or in other words, to single-step the reduction. Again, this can be achieved using side effects in the meta-language Standard ML. We simply make the context function `c` count the number of redexes contracted (substitutions performed), and set a step limit N before evaluation is started.

When N redexes have been contracted, `c` aborts the reduction by raising an exception `Enough` e', which carries as its argument the term e' that had been obtained when reaching the limit. An enclosing exception handler handles this exception and reports e' as the result of the reduction. The next invocation of the reduction function simply sets the step limit N one higher, and so on. Thus the reduction of the original term starts over for every new step, but we create the illusion of reducing the term one step at a time.

The main drawback of this approach is that the total time spent performing n steps of reduction is $O(n^2)$. In practice, this does not matter: one does not care to single-step very long computations.

10 Conclusion

We have described a simple way to implement lambda calculus reduction, describing reduction strategies using big-step operational semantics, implementing reduction by straightforward reduction functions in Standard ML, and instrumenting them to produce a trace of the reduction, using contexts. This approach is easily extended to other reduction strategies describable by big-step operational semantics. The extension to lazy evaluation, whether using graph reduction or an explicit heap, would be complicated mostly because of the need to print the current graph or heap.

The functions for reduction in context were useful for creating a web interface also, running the reduction functions as a CGI script written in ML. The web interface provides a simple platform for students' experiments with lambda calculus encodings and reduction strategies.

References

[1] Augustsson, L., *A Compiler for Lazy ML*, in 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, 218–227.

[2] Barendregt, H.P., "The Lambda Calculus. Its Syntax and Semantics", North-Holland 1984.

[3] Church, A., *A Note on the Entscheidungsproblem*, Journal of Symbolic Logic **1** (1936) 40–41, 101–102.

[4] Kahn, G., *Natural Semantics*, in STACS 87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany. (Lecture Notes in Computer Science, vol. 247, Springer-Verlag 1987, 22–39.

[5] Launchbury, J., *A Natural Semantics for Lazy Evaluation*, in Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993, 144–154.

[6] Milner, R., M. Tofte, R. Harper and D.B. MacQueen, "The Definition of Standard ML (Revised)", The MIT Press 1997.

[7] Moscow ML is available at `http://www.dina.kvl.dk/~sestoft/mosml.html`

[8] Paulson, L.C., "ML for the Working Programmer", second edition, Cambridge University Press 1996.

[9] Peyton Jones, S.L., "The Implementation of Functional Programming Languages", Prentice-Hall 1987.

[10] Peyton Jones, S.L. and J. Hughes (editors): "Haskell 98: A Non-Strict, Purely Functional Language", 1999, at `http://www.haskell.org/onlinereport/`

[11] "Revised^4 Report on the Algorithmic Language Scheme", IEEE Std 1178-1990, Institute of Electrical and Electronic Engineers 1991.

[12] Sestoft, P., *Deriving a lazy abstract machine*, Journal of Functional Programming **7**, 3 (1997) 231–264.