Demonstrating Lambda Calculus Reduction

Peter Sestoft

Department of Mathematics and Physics
Royal Veterinary and Agricultural University, Denmark
and
IT University of Copenhagen, Denmark
`sestoft@dina.kvl.dk`

Abstract. We describe lambda calculus reduction strategies, such as call-by-value, call-by-name, normal order, and applicative order, using big-step operational semantics. We show how to simply and efficiently trace such reductions, and use this in a web-based lambda calculus reducer available at ⟨http://www.dina.kvl.dk/˜sestoft/lamreduce/⟩.

1 Introduction

The pure untyped lambda calculus is often taught as part of the computer science curriculum. It may be taught in a computability course as a classical computation model. 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 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:

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

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 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.

To do this we also precisely define and clarify 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. These have been given a number of different interpretations in the literature.

2 Motivation and Related Work

Much has been written about the lambda calculus since Church developed it as a foundation for mathematics [6]. Landin defined the semantics of programming languages in terms of the lambda calculus [11], and gave a call-by-value interpreter for it: the SECD-machine [10]. Strachey used lambda calculus as a meta-language for denotational semantics, and Scott gave models for the pure untyped lambda calculus, making sure that self-application could be assigned a meaning; see Stoy [22]. Self-application $(x\,x)$ of a term x is used when encoding recursion, for instance in Church's Y combinator:

$$Y \equiv \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x)) \tag{2}$$

Plotkin studied the call-by-value lambda calculus corresponding to the functional language ISWIM [12] implemented by Landin's SECD-machine, and also a related call-by-name lambda calculus, and observed that one characteristic of a functional programming language was the absence of reduction under lambda abstractions [19].

Barendregt [4] is the standard reference on the untyped lambda calculus, with emphasis on models and proof theory, not programming languages.

Many textbooks on functional programming or denotational semantics present the pure untyped lambda calculus, show how to encode numbers and algebraic data types, and define evaluators for it. One example is Paulson's ML textbook [16], which gives interpreters for call-by-name as well as call-by-value.

So is there really a need for yet another paper on lambda calculus reduction? We do think so, because it is customary to look at the lambda calculus either from the programming language side or from the calculus or model side, leaving the relations between the sides somewhat unclear.

For example, Plotkin [19] defines call-by-value reduction as well as call-by-name reduction, but the call-by-name rules take free variables into account only to a limited extent. By the rules, $x\,((\lambda z.z)\,v)$ reduces to $x\,v$, but $(x\,y)\,((\lambda z.z)\,v)$ does not reduce to $x\,y\,v$ [19, page 146]. Similarly, the call-by-value strategy described by Felleisen and Hieb using evaluation contexts [8, Section 2] would not reduce $(x\,y)\,((\lambda z.z)\,v)$ to $x\,y\,v$, since there is no evaluation context of the form $(x\,y)\,[\,]$. This is unproblematic because, following Landin, these researchers were interested only in terms with no free variables, and in reduction only outside lambda abstractions.

But it means that the reduction rules are not immediately useful for terms that have free variables, and therefore not useful for experimentation with the terms that result from encoding programming language constructs in the pure lambda calculus.

Conversely, Paulson [16] presents call-by-value and call-by-name interpreters for the pure lambda calculus that do handle free variables. However, they also perform reduction under lambda abstractions (unlike functional programming languages), and the evaluation order is not leftmost outermost: under call-by-name, an application $(e_1\,e_2)$ is reduced by first reducing e_1 to head normal form, so redexes inside e_1 may be contracted before an enclosing leftmost redex. This makes the relation between Paulson's call-by-name and normal order (leftmost outermost) reduction strategies somewhat unclear.

Therefore we find that it may be useful to contrast the various reduction strategies, present them using big-step operational semantics, present their (naive) implementation in ML, and show how to obtain a trace of the reduction.

3 The Pure Untyped Lambda Calculus

We use the pure untyped lambda calculus [4]. A lambda term is a *variable* x, a lambda *abstraction* $\lambda x.e$ which binds x in e, or an *application* $(e_1\, e_2)$ of a 'function' e_1 to an 'argument' e_2:

$$e ::= \; x \mid \lambda x.e \mid e_1\, e_2 \tag{3}$$

Application associates to the left, so $(e_1\, e_2\, e_3)$ means $((e_1\, e_2)\, e_3)$. A lambda term 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. The notation $e[e_x/x]$ denotes substitution of e_x for x in e, with renaming of bound variables in e if necessary to avoid capture of free variables in e_x.

A *redex* is a subterm of the form $((\lambda x.e)\, e_2)$; the *contraction* of a redex produces $e[e_2/x]$, substituting the argument e_2 for every occurrence of the parameter x in e. By $e \longrightarrow_\beta e'$ we denote *β-reduction*, the contraction of some redex in e to obtain e'.

A redex is to the *left* of another redex if its lambda abstractor appears further to the left. The *leftmost outermost* redex (if any) is the leftmost redex not contained in any other redex. The *leftmost innermost* redex (if any) is the leftmost redex not containing any other redex.

4 Functional Programming Languages

In practical functional programming languages such as Scheme [20], Standard ML [14] or Haskell [18], programs cannot have free variables, and reductions are not performed under lambda abstractions or other variable binders, because this would considerably complicate their efficient implementation [17].

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 x 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 terms 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. Section 6 define these normal forms.

5 Lazy Functional Programming Languages

Under lazy evaluation, a variable-bound term is evaluated at most once, regardless how often the variable is used [17]. Thus an argument term may not be duplicated before it has been reduced, and may be reduced only if actually used. This evaluation mechanism may be called call-by-need, or call-by-name with sharing of argument evaluation. The equational theory of call-by-need lambda calculus has been studied by Ariola and Felleisen [2] among others. (By contrast, the lazy lambda calculus of Abramsky and Ong [1] is not lazy in the sense discussed here; rather, it is the theory of call-by-name lambda calculus, without reduction under abstractions.)

Lazy functional languages also permit the creation of cyclic terms, or cycles in the heap. For instance, this declaration creates a finite (cyclic) representation of an infinite list of 1's:

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

Thus to be true also to the intensional properties of lazy languages (such as time and space consumption), a model should be able to describe such constant-size cyclic structures. Substitution of terms for variables cannot truly model them, only approximate them by unfolding of a recursive term definition, possibly encoded using a recursion combinator such as (2). To properly express sharing of subterm evaluation, and the creation of cyclic terms, one must extend the syntax (3) with mutually recursive bindings:

$$e ::= \; x \mid \lambda x.e \mid e\, e \mid \mathit{letrec}\ \{x_i = e_i\}\ \mathit{in}\ e \tag{4}$$

The sharing of subterm evaluation and the dynamic creation of cyclic terms may be modelled using graph reduction, as suggested by Wadsworth [24] and used in subsequent work [3,17,23], or using an explicit heap [13,21].

Thus a proper modelling of lazy evaluation, with sharing of argument evaluation and cyclic data structures, requires syntactic extensions as well as a more elaborate evaluation model than just term rewriting. We shall not consider lazy evaluation any further in this paper, and shall consider only the syntax in (3) above.

6 Normal Forms

We need to distinguish four different normal forms, depending on whether we reduce under abstractions 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).

Figure 1 summarizes the four normal forms using four context-free grammars. In each grammar, the symbol E denotes a term in the relevant normal form, e denotes an arbitrary lambda term generated by (3), and $n \geq 0$. Note how the two dichotomies generate the four normal forms just by varying the form of lambda abstraction bodies and application arguments.

	Reduce under abstractions	
Reduce args	**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 $E ::= \lambda x.e \mid x\, e_1 \ldots e_n$

Fig. 1. Normal forms. The e_i denote arbitrary lambda terms generated by (3).

7 Reduction Strategies and Reduction Functions

We present several reduction strategies using big-step operational semantics, or natural semantics [9], and their implementation in Standard ML. The premises of each semantic rule are assumed to be evaluated from left to right, although this is immaterial to their logical interpretation. We exploit that Standard ML evaluates a function's arguments before calling the function, evaluates the right-hand side of `let`-bindings before binding the variable, and evaluates subterms from left to right [14].

We model lambda terms x, $\lambda x.e$ and $(e\, e)$ as ML constructed data, representing variable names 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 contracting the redex $(\lambda x.e)\, e_x$.

7.1 Call-by-Name Reduction to Weak Head Normal Form

Call-by-name reduction $e \xrightarrow{bn} e'$ reduces the leftmost outermost redex not inside a lambda abstraction first. It treats free variables as non-strict data constructors.

For terms without free variables, it coincides with Plotkin's call-by-name reduction [19, Section 5], and is closely related to Engelfriet and Schmidt's outside-in derivation (in context-free tree grammars, or first-order recursion equations) [7, page 334].

$$x \xrightarrow{bn} x$$

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

$$\frac{e_1 \xrightarrow{bn} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{bn} e'}{(e_1\ e_2) \xrightarrow{bn} e'} \tag{5}$$

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

It is easy to see that all four rules generate terms in weak head normal form. In particular, in the last rule e_1' must have form $y\, e_{11}' \ldots e_{1n}'$ for some $n \geq 0$, so $(e_1'\, e_2)$ is a weak head normal form. Assuming that the rule premises are read and 'executed' from left to right, it is also clear that only leftmost redexes are contracted. No reduction is performed under abstractions.

The following ML function `cbn` computes the weak head normal form of a lambda term, contracting redexes in the order implied by the operational semantics (5) above:

```
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)
```

The first function clause above handles variables x and implements the first semantics rule. Similarly, the second function clause handles lambda abstractions $(\lambda x.e)$ and implements the second semantics rule. In both cases, the given term is returned unmodified. The third function clause handles applications $(e_1\, e_2)$ and implements the third and fourth semantics rule by discriminating on the result of reducing e_1. If the result is a lambda abstraction $(\lambda x.e)$ then the `cbn` function is called to reduce the expression $e[e_2/x]$; but if the result is any other expression e_1', the application $(e_1'\, e_2)$ is returned.

In all cases, this is exactly what the semantics rules in (5) describe. In fact, one can see that $e \xrightarrow{bn} e'$ if and only if `cbn` e terminates and returns e'.

7.2 Normal Order Reduction to Normal Form

Normal order reduction $e \xrightarrow{no} e'$ reduces the leftmost outermost redex first. In an application $(e_1\, e_2)$ the function term e_1 must be reduced using call-by-name (5).

Namely, if e_1 reduces to an abstraction $(\lambda x.e)$, then the redex $((\lambda x.e)\, e_2)$ must be reduced before redexes in e, if any, because they would not be outermost.

$$
\begin{array}{c}
x \xrightarrow{no} x \\[2ex]
\dfrac{e \xrightarrow{no} e'}{(\lambda x.e) \xrightarrow{no} (\lambda x.e')} \\[2ex]
\dfrac{e_1 \xrightarrow{bn} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{no} e'}{(e_1\ e_2) \xrightarrow{no} e'} \\[2ex]
\dfrac{e_1 \xrightarrow{bn} e'_1 \not\equiv (\lambda x.e) \qquad e'_1 \xrightarrow{no} e''_1 \qquad e_2 \xrightarrow{no} e'_2}{(e_1\ e_2) \xrightarrow{no} (e''_1\ e'_2)}
\end{array}
\tag{6}
$$

It is easy to see that these rules generate normal form terms only. In particular, in the last rule e'_1 must have form $y\, e'_{11} \ldots e'_{1n}$ for some $n \geq 0$, so e''_1 must have form $y\, E''_{11} \ldots E''_{1n}$ for some normal forms E''_{1i}, and therefore $(e''_1\, e'_2)$ is a normal form. Any redex contracted is the leftmost one not contained in any other redex; this relies on the use of call-by-name in the application rules. Reductions are performed also under lambda abstractions. Normal order reduction is *normalizing*: if the term e has a normal form, then normal order reduction of e will terminate (with the normal form as result).

The Standard ML function `nor : lam -> lam` below implements the reduction strategy. Note that it uses the function `cbn` defined in Section 7.1:

```
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
```

Again the first two cases of the function implement the first two reduction rules. The third case implements the third and fourth rules by evaluating e_1 using call-by-name `cbn` and then discriminating on whether the result is a lambda abstraction or not, as in the third and fourth rule in (6).

7.3 Call-by-Value Reduction to Weak Normal Form

Call-by-value reduction $e \xrightarrow{bv} e'$ reduces the leftmost innermost redex not inside a lambda abstraction first. It treats free variables as strict data constructors. For terms without free variables, it coincides with call-by-value reduction as defined by Plotkin [19, Section 4] and Felleisen and Hieb [8]. It is closely related to

Engelfriet and Schmidt's inside-out derivations (in context-free tree grammars, or first-order recursion equations) [7, page 334]. It differs from call-by-name (Section 7.1) only by reducing the argument e_2 of an application $(e_1\, e_2)$ before contracting the redex, and before building an application term:

$$
\begin{gathered}
x \xrightarrow{bv} x \\[1em]
(\lambda x.e) \xrightarrow{bv} (\lambda x.e) \\[1em]
\frac{e_1 \xrightarrow{bv} (\lambda x.e) \qquad e_2 \xrightarrow{bv} e_2' \qquad e[e_2'/x] \xrightarrow{bv} e'}{(e_1\ e_2) \xrightarrow{bv} e'} \\[1em]
\frac{e_1 \xrightarrow{bv} e_1' \not\equiv (\lambda x.e) \qquad e_2 \xrightarrow{bv} e_2'}{(e_1\ e_2) \xrightarrow{bv} (e_1'\ e_2')}
\end{gathered}
\tag{7}
$$

It is easy to see that these rules generate weak normal form terms only. In particular, in the last rule e_1' must have form $y\, E_{11}' \, \ldots \, E_{1n}'$ for some $n \geq 0$ and weak normal forms E_{1i}', and therefore $(e_1'\, e_2')$ is a weak normal form too. No reductions are performed under lambda abstractions. This is Paulson's `eval` auxiliary function [16, page 390]. The implementation of the rules by an ML function is straightforward and is omitted.

7.4 Applicative Order Reduction to Normal Form

Applicative order reduction $e \xrightarrow{ao} e'$ reduces the leftmost innermost redex first. It differs from call-by-value (Section 7.3) only by reducing also under abstractions:

$$
\begin{gathered}
x \xrightarrow{ao} x \\[1em]
\frac{e \xrightarrow{ao} e'}{(\lambda x.e) \xrightarrow{ao} (\lambda x.e')} \\[1em]
\frac{e_1 \xrightarrow{ao} (\lambda x.e) \qquad e_2 \xrightarrow{ao} e_2' \qquad e[e_2'/x] \xrightarrow{ao} e'}{(e_1\ e_2) \xrightarrow{ao} e'} \\[1em]
\frac{e_1 \xrightarrow{ao} e_1' \not\equiv (\lambda x.e) \qquad e_2 \xrightarrow{ao} e_2'}{(e_1\ e_2) \xrightarrow{ao} (e_1'\ e_2')}
\end{gathered}
\tag{8}
$$

It is easy to see that the rules generate only normal form terms. As before, note that in the last rule e_1' must have form $y\, E_{11}' \, \ldots \, E_{1n}'$ for some $n \geq 0$ and normal forms E_{1i}'. Also, it is clear that when a redex $((\lambda x.e)\, e_2')$ is contracted, it contains no other redex, and it is the leftmost redex with this property.

Applicative order reduction is not normalizing; with $\Omega \equiv (\lambda x.(x\,x))(\lambda x.(x\,x))$ it produces an infinite reduction $((\lambda x.y)\,\Omega) \longrightarrow_\beta ((\lambda x.y)\,\Omega) \longrightarrow_\beta \ldots$ although the term has normal form y.

In fact, applicative order reduction fails to normalize applications of functions defined using recursion combinators, even with recursion combinators designed for call-by-value, such as Y_v:

$$Y_v \equiv \lambda h.(\lambda x.\lambda a.h\,(x\,x)\,a)\,(\lambda x.\lambda a.h\,(x\,x)\,a) \tag{9}$$

7.5 Hybrid Applicative Order Reduction to Normal Form

Hybrid applicative order reduction is a hybrid of call-by-value and applicative order reduction. It reduces to normal form, but reduces under lambda abstractions only in argument positions. Therefore the usual call-by-value versions of the recursion combinator, such as (9) above, may be used with this reduction strategy. Thus the hybrid applicative order strategy normalizes more terms than applicative order reduction, while using fewer reduction steps than normal order reduction. The hybrid applicative order strategy relates to call-by-value in the same way that the normal order strategy relates to call-by-name. It resembles Paulson's call-by-value strategy, which works in two phases: first reduce the term by $\xrightarrow{bv}$, then normalize the bodies of any remaining lambda abstractions [16, page 391].

$$\begin{gathered}
x \xrightarrow{ha} x \\[2ex]
\frac{e \xrightarrow{ha} e'}{(\lambda x.e) \xrightarrow{ha} (\lambda x.e')} \\[2ex]
\frac{e_1 \xrightarrow{bv} (\lambda x.e) \qquad e_2 \xrightarrow{ha} e_2' \qquad e[e_2'/x] \xrightarrow{ha} e'}{(e_1\ e_2) \xrightarrow{ha} e'} \\[2ex]
\frac{e_1 \xrightarrow{bv} e_1' \not\equiv (\lambda x.e) \qquad e_1' \xrightarrow{ha} e_1'' \qquad e_2 \xrightarrow{ha} e_2'}{(e_1\ e_2) \xrightarrow{ha} (e_1''\ e_2')}
\end{gathered} \tag{10}$$

7.6 Head Spine Reduction to Head Normal Form

The head spine strategy performs reductions inside lambda abstractions, but only in head position. This is the reduction strategy implemented by Paulson's `headNF` function [16, page 390].

$$\begin{gathered}
x \xrightarrow{he} x \\[1em]
\frac{e \xrightarrow{he} e'}{(\lambda x.e) \xrightarrow{he} (\lambda x.e')} \\[1em]
\frac{e_1 \xrightarrow{he} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{he} e'}{(e_1\ e_2) \xrightarrow{he} e'} \\[1em]
\frac{e_1 \xrightarrow{he} e_1' \not\equiv (\lambda x.e)}{(e_1\ e_2) \xrightarrow{he} (e_1'\ e_2)}
\end{gathered} \tag{11}$$

It is easy to see that the rules generate only head normal form terms. Note that this is not a *head reduction* as defined by Barendregt [4, Definition 8.3.10]: In a (leftmost) head reduction only head redexes are contracted, where a redex $((\lambda x.e_0)\, e_1)$ is a *head redex* if it is preceded to the left only by lambda abstractors of non-redexes, as in $\lambda x_1. \ldots \lambda x_n.(\lambda x.e_0)\, e_1\ \ldots\ e_m$, with $n \geq 0$ and $m \geq 1$.

To define head reduction, one should use $e_1 \xrightarrow{bn} e_1'$ in the above application rules (11) to avoid premature reduction of inner redexes, similar to the use of $\xrightarrow{bn}$ in the definition of $\xrightarrow{no}$.

7.7 Hybrid Normal Order Reduction to Normal Form

Hybrid normal order reduction is a hybrid of head spine reduction and normal order reduction. It differs from normal order reduction only by reducing the function e_1 in an application to head normal form (by $\xrightarrow{he}$) instead of weak head normal form (by $\xrightarrow{bn}$) before applying it to the argument e_2.

The hybrid normal order strategy resembles Paulson's call-by-name strategy, which works in two phases: first reduce the term by $\xrightarrow{he}$ to head normal form, then normalize unevaluated arguments and bodies of any remaining lambda abstractions [16, page 391].

$$\begin{gathered}
x \xrightarrow{hn} x \\[1em]
\frac{e \xrightarrow{hn} e'}{(\lambda x.e) \xrightarrow{hn} (\lambda x.e')} \\[1em]
\frac{e_1 \xrightarrow{he} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{hn} e'}{(e_1\ e_2) \xrightarrow{hn} e'} \\[1em]
\frac{e_1 \xrightarrow{he} e_1' \not\equiv (\lambda x.e) \qquad e_1' \xrightarrow{hn} e_1'' \qquad e_2 \xrightarrow{hn} e_2'}{(e_1\ e_2) \xrightarrow{hn} (e_1''\ e_2')}
\end{gathered} \tag{12}$$

These rules generate normal form terms only. The strategy is normalizing, because if the term $(e_1\, e_2)$ has a normal form, then it has a head normal form, and then so has e_1 [4, Proposition 8.3.13].

8 Properties of the Reduction Strategies

The relation defined by each reduction strategy is idempotent. For instance, if $e \xrightarrow{bn} e'$ then $e' \xrightarrow{bn} e'$. To see this, observe that e' is in weak head normal form, so it has form $\lambda x.e''$ or $x\, e_1\, \ldots\, e_n$, where e'' and $e_1, \ldots, e_n$ are arbitrary lambda terms. In the first case, e' reduces to itself by the second rule of (5). In the second case, an induction on n shows that e' reduces to itself by the first and third rule of (5). Similar arguments can be made for the other reduction strategies.

Figure 2 classifies the seven reduction strategies presented in Sections 7.1 to 7.7 according the normal forms (Figure 1) they produce.

	Reduce under abstractions	
Reduce args	**Yes**	**No**
Yes	Normal form **ao**, *no*, *ha*, *ho*	Weak normal form **bv**
No	Head normal form **he**	Weak head normal form **bn**

Fig. 2. Classification of reduction strategies by the normal forms they produce. The 'uniform' reduction strategies are shown in boldface, the 'hybrid' ones in italics.

Inspection of the big-step semantics rules shows that four of the reduction strategies (**ao**, **bn**, **bv**, **he**, shown in bold in Figure 2) are 'uniform': their definition involves only that reduction strategy itself. The remaining three (*no*, *ha*, *hn*) are 'hybrid': each uses one of the 'uniform' strategies for the reduction of the expression e_1 in function position in applications $(e_1\, e_2)$. Figure 3 shows how the 'hybrid' and 'uniform' strategies are related.

Hybrid	**Uniform**
no	**bn**
ha	**bv**
hn	**he**

Fig. 3. Derivation of hybrid strategies from uniform ones.

9 Tracing: Side-Effecting Substitution, and Contexts

The reducers defined in ML in Section 7 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 $((\lambda x.e)\, e_x)$, and its *result* is the result of the substitution $e[e_x/x]$, which is the contracted redex.

We could define a function `printlam : lam -> unit` that prints the given lambda term as a side effect. Then replacing the call `subst e2 (Lam(x, e))` in function `cbn` of Section 7.1 by `csubst printlam e2 (Lam(x, e))` will cause the reduction of a term by `cbn` to produce a printed trace of all 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 printed redex occurs. This is because the function `printlam` is applied only to the redex itself; the term surrounding the redex is implicit. To make the term surrounding the redex explicit, we can use a *context*, a term with a single hole, such as $\lambda x.[\,]$ or $(e_1\, [\,])$ or $([\,]\, e_2)$, where the hole is denoted by $[\,]$. Filling the hole of a context with a lambda term produces a lambda term. The following grammar generates all single-hole contexts:

$$C \;::=\; [\,] \mid \lambda x.C \mid e\, C \mid C\, e \tag{13}$$

A context can be represented by an ML function of type `lam -> lam`. The four forms of contexts (13) can be created 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 the term $(e_1\, e_2)$.

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 $((\lambda x.[\,])\, e_2)$.

10 Reduction in Context

To produce a trace of the reduction, we modify the reduction functions defined in Section 7 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 7.1) 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)
```

The trace of a reduction can be defined also by direct instrumentation of the operational semantics (5). Let us define a *trace* to be a finite sequence of lambda terms, denote the empty trace by ϵ, and denote the concatenation of traces s and t by $s \cdot t$. Now we define the relation $e \overset{s}{\xrightarrow{bn}_{C}} e'$ to mean: under call-by-name, the expression e reduces to e', and if e appears in context C, then s is the trace of the reduction. The trace s will be empty if no redex was contracted in the reduction. If some redex was contracted, the first term in the trace will be e.

The tracing relation corresponding to call-by-name reduction (5) can be defined as shown below:

$$
\begin{gathered}
x \xrightarrow{bn}{}_{C}^{\epsilon}\ x \\[1ex]
(\lambda x.e) \xrightarrow{bn}{}_{C}^{\epsilon}\ (\lambda x.e) \\[1ex]
\frac{e_1 \xrightarrow{bn}{}_{C[[\,]\,e_2]}^{s}\ (\lambda x.e) \qquad e[e_2/x] \xrightarrow{bn}{}_{C}^{t}\ e'}{(e_1\ e_2) \xrightarrow{bn}{}_{C}^{s\cdot C[(\lambda x.e)\,e_2]\cdot t}\ e'} \\[1ex]
\frac{e_1 \xrightarrow{bn}{}_{C[[\,]\,e_2]}^{s}\ e_1' \not\equiv \lambda x.e}{(e_1\ e_2) \xrightarrow{bn}{}_{C}^{s}\ (e_1'\ e_2)}
\end{gathered}
\tag{14}
$$

Thus reduction of a variable x or a lambda abstraction $(\lambda x.e)$ produces the empty trace ϵ. When e_1 reduces to a lambda abstraction, reduction of the application $(e_1\,e_2)$ produces the trace $s \cdot C[(\lambda x.e)\,e_2] \cdot t$, where s traces the reduction of e_1 and t traces the reduction of the contracted redex $e[e_2/x]$.

Tracing versions of the other reduction strategies can be defined analogously.

11 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 implementation language. 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 after N reductions. An enclosing exception handler returns 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: noboby wants to single-step very long computations.

12 A Web-Based Interface to the Reduction Functions

A web-based interface to the tracing reduction functions can be implemented as an ordinary CGI script. The lambda term to be reduced, the name of the desired reduction strategy, the kind of computation (tracing, single-stepping, etc.) and the step limit are passed as parameters to the script.

Such an implementation has been written in Moscow ML [15] and is available at ⟨http://www.dina.kvl.dk/~sestoft/lamreduce/⟩. The implementation uses the `Mosmlcgi` library to access CGI parameters, and the `Msp` library for efficient structured generation of HTML code.

For tracing, the script uses a function `htmllam : lam -> unit` that prints a lambda term as HTML code, which is then sent to the browser by the web server. Calling `cbnc` (or any other tracing reduction function) with `htmllam` as argument will display a trace of the reduction in the browser.

A trick is used to make the next redex into a hyperlink in the browser. The implementation's representation of lambda terms is extended with labelled subterms, and `csubst` attaches labels $0, 1, \ldots$ to redexes in the order in which they are contracted. When single-stepping a reduction, the last labelled redex inside the term can be formatted as a hyperlink. Clicking on the hyperlink will call the CGI script again to perform one more step of reduction, creating the illusion of single-stepping the reduction as explained above.

13 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.

We find that big-step semantics provides a clear presentation of the reduction strategies, highlighting their differences and making it easy to see what normal forms they produce.

The extension to lazy evaluation, whether using graph reduction or an explicit heap, would be complicated mostly by the need to represent the current term graph or heap, and to print it in a comprehensible way.

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.

Acknowledgements Thanks for Dave Schmidt for helpful comments that have improved contents as well as presentation.

References

1. Abramsky, S., Ong, C.-H.L.: Full Abstraction in the Lazy λ-Calculus. Information and Computation **105**, 2 (1993) 159–268.

2. Ariola, Z.M., Felleisen, M.: The Call-by-Need Lambda Calculus. Journal of Functional Programming **7**, 3 (1997) 265–301.
3. Augustsson, L.: A Compiler for Lazy ML. In: 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas. ACM Press (1984) 218–227.
4. Barendregt, H.P.: The Lambda Calculus. Its Syntax and Semantics. North-Holland (1984).
5. Barendregt, H.P. *et al.*: Needed Reduction and Spine Strategies for the Lambda Calculus. Information and Computation **75** (1987) 191–231.
6. Church, A.: A Note on the Entscheidungsproblem. Journal of Symbolic Logic **1** (1936) 40–41, 101–102.
7. Engelfriet, J., Schmidt, E.M.: IO and OI. Journal of Computer and System Sciences **15** (1977) 328–353 and **16** (1978) 67–99.
8. Felleisen, M., Hieb, R.: The Revised Report on the Syntactic Theories of Sequential Control and State. Theoretical Computer Science **103**, 2 (1992) 235–271.
9. 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.
10. Landin, P.J.: The Mechanical Evaluation of Expressions. Computer Journal **6**, 4 (1964) 308–320.
11. Landin, P.J.: A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I. Communications of the ACM **8**, 2 (1965) 89–101.
12. Landin, P.J.: The Next 700 Programming Languages. Communications of the ACM **9**, 3 (1966) 157–166.
13. Launchbury, J.: A Natural Semantics for Lazy Evaluation. In: Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993. ACM Press (1993) 144–154.
14. Milner, R., Tofte, M., Harper, R., MacQueen, D.B.: The Definition of Standard ML (Revised). The MIT Press (1997).
15. Moscow ML is available at ⟨http://www.dina.kvl.dk/~sestoft/mosml.html⟩.
16. Paulson, L.C.: ML for the Working Programmer. Second edition. Cambridge University Press (1996).
17. Peyton Jones, S.L.: The Implementation of Functional Programming Languages. Prentice-Hall (1987).
18. Peyton Jones, S.L., Hughes, J. (eds.): Haskell 98: A Non-Strict, Purely Functional Language. At ⟨http://www.haskell.org/onlinereport/⟩.
19. Plotkin, G.: Call-by-Name, Call-by-Value and the λ-Calculus. Theoretical Computer Science **1** (1975) 125–159.
20. Revised^4 Report on the Algorithmic Language Scheme, IEEE Std 1178-1990. Institute of Electrical and Electronic Engineers (1991).
21. Sestoft, P.: Deriving a Lazy Abstract Machine. Journal of Functional Programming **7**, 3 (1997) 231–264.
22. Stoy, J.E.: The Scott-Strachey Approach to Programming Language Theory. The MIT Press (1977).
23. Turner, D.A.: A New Implementation Technique for Applicative Languages. Software – Practice and Experience **9** (1979) 31–49.
24. Wadsworth, C.P.: Semantics and Pragmatics of the Lambda Calculus. D.Phil. thesis, Oxford University, September 1971.