COMPILING LAMBDA CALCULUS

Nils M Holm

Contents

Preface

Lambda calculus is an abstract term rewriting system that was originally designed to solve mathematical problems on a sheet of paper. Paper is an inherently *immutable* tool, though: each term is rewritten by placing a modified term under the original one.

Computer systems, on the other hand, are inherently mutable systems: problems are solved by changing the values of registers.

This book describes how to adapt lambda calculus to computer-based environments, by interpreting it or translating it to a notation that is better fitted for processing by a register-based machine.

Chapter one of the book describes lambda calculus in general: its syntax, composition of formulae, common abbreviations, and the rules that are used to *reduce* (i.e. compute) lambda terms. This chapter introduces the basic theory of lambda calculus.

The second chapter illustrates how lambda calculus works in practice, by applying it to fundamental problems like the implementation of natural numbers, ordered pairs, and lists. It also introduces fixed points and recursive formulae.

The third chapter implements a very simple subset of the Scheme language, which is loosely based on lambda calculus.

It starts with a simple interpreter for a tiny subset language and then modifies it subsequently in order to cover a wider range of the Scheme language and finally shifts the focus from interpretation to compilation and code generation. The full code for all interpreters and compilers is included in the text.

The concluding chapter presents a language that resembles lambda calculus more closely, by implementing currying and partial evaluation. The implications of the design are outlined and compared to typical LISP and Scheme implementations. Again, a working compiler for the language is included in the text.

This book is intended for compiler writers and people who are interested in lambda calculus and LISPy languages. It focuses on theory as well as interpretation and code generation.

The code in the book is mostly in lambda calculus and Scheme, with a few short passages in C for addressing the more technical side of program interpretation (i.e. runtime support and code generation).

Machine-readable versions of the programs in this book may be found on the author’s homepage: **`http://t3x.org`** .

Although the code in the book is simple and straight-forward, some familiarity with Scheme and C is certainly beneficial to following the text, especially the later chapters.

And now, as usual, enjoy the tour!

Nils M Holm, Nov 2016

Second Edition

The first edition of “Compiling Lambda Calculus” was published in a rush, so it accumulated a lot of errata in the months after its publication. In this new edition a lot of small mistakes have been corrected and the prose has been revised in some places.

The section on optimization in the “LC-based language” chapter now contains an algorithm for environment pruning, which was only outlined in the first edition, because other matters got in the way. I am glad that I finally found the time to get back to this book, pick up some lose ends, and add the finished touches.

Nils M Holm, Jul 2018

Lambda Calculus

Lambda calculus (or λ-calculus) forms the theoretical basis for most functional programming languages. It is often associated with concepts like “anonymous functions” or “closures” but we will start from the beginning here and have a closer look at the formal system that is λ-calculus.

A *formal system* is a mathematical concept for manipulating terms of a formal language. It is is comprised of three parts:

- notation (syntax)
- theory (axioms and rules)
- semantics (meaning of terms)

In the following, the notation and theory of λ-calculus will be given formally and semantics will be specified informally as required.

Notation

The *alphabet* of λ-calculus (LC) is the following:

$\lambda \;\; (\;\;) \;\; a_i \cdots z_i \;\; A_i \cdots Z_i$

where $a \cdots z$ denotes the letters from a to z, a_i denotes the letters $a_0, a_1, \cdots$, etc. Any letter of the form x_0 is normally written x. There is an infinite number of letters. Some texts use $\bar{a}, \bar{\bar{a}}$, etc instead of $a_1, a_2, \cdots$.

By convention, lower case letters (x, y, ...) represent variables and upper case letters (M, N, ...) represent terms, which may in turn be variables, abstractions, or applications (see below).

A *well-formed formula* (*wff*) is a any combination of symbols from the alphabet of a formal system that forms a correct term under the rules of that system. The wffs of LC are defined inductively as follows [Church1941]:

(W1) $x \in \{a_i, \cdots, z_i, A_i, \cdots, Z_i\}$ is a wff. (variable)

(W2) If x is a variable and M is a wff, $(\lambda x M)$ is a wff. (abstraction)

(W3) if M and N are wffs, (MN) is a wff. (application)

Only combinations of symbols that can be built from the above rules are valid terms of the LC. For instance:

$x \quad (\lambda xx) \quad (MN) \quad (x(yz)) \quad ((\lambda x(xx))y) \quad (\lambda a(\lambda b((Fb)a)))$

A *variable* is similar to a variable in a programming language, with one minor difference: a variable does not necessarily have a value, but it can stand for itself. In other words: LC does not assign any special meaning to a variable other than its lexical form. Variables may be acted upon by the rules outlined later in this chapter.

Abstraction can be thought of as the creation of a new *function* $(\lambda x M)$, where x is the independent variable of the function and M is its term. In pure λ-calculus, x must appear in M, while in λK-calculus (pg 16), it does not have to. This text deals with λK-calculus exclusively.

(FN) denotes the *application* of the term F to the term N. If F is an abstraction of the form $(\lambda x M)$, then

(App) $(FN) = ((\lambda x M)N) = M[x{:=}N]$

where $M[x{:=}N]$ denotes M with all occurrences of x replaced by N, i.e. the *substitution* of N for x in M. For example:

$((\lambda x(\lambda y(yx)))A) \; = \; (\lambda y(yx))[x{:=}A] \; = \; (\lambda y(yA))$

The F in (FN) is called the *function* and N is called the *argument* in the context of application.

If M in (MN) is not an abstraction, then (MN) is in its “normal form” (pg 14) and the term is not rewritten in any way.

Unfortunately, application is a bit more complex than outlined here. We will return to it shortly (pg 11).

Syntax and Short Cuts

Because λ-terms can become pretty unwieldy pretty quickly, the *abbreviations* (A1) through (A5), listed in figure 1, are commonly used.

(A1)	(λxM)	λxM	
(A2)	(MN)	MN	
(A3)	$((MN)P)$	MNP	
(A4)	$(M(NP))$	$M(NP)$	
(A5)	$(\lambda x(\lambda yM))$	$\lambda xy(M)$	also: $\lambda x\lambda y(M)$

Figure 1 - Abbreviations

Furthermore, a dot is used to bracket everything to the right of the dot, e.g.:

$$\lambda x.\, yz \;\equiv\; \lambda x(yz) \;\equiv\; (\lambda x(yz))$$
$$\lambda xyz.\, xz(yz) \;\equiv\; \lambda xyz(xz(yz)) \;\equiv\; (\lambda x(\lambda y(\lambda z((xz)(yz)))))$$

The notation $x \equiv y$ is used to indicate that the terms x and y are syntactically *equivalent*. The notation is also used to *define* new functions, by assigning names to terms.

As can be seen from (A3), application *associates* to the left, i.e. in a sequence of two operations, the left one will take precedence over the right one: $MNP = (MN)P$.

Abstraction, on the other hand, associates to the right (A5):

$$\lambda xyz.\, xz(yz) \;=\; \lambda x(\lambda y(\lambda z.\, xz(yz)))$$

This follows immediately from definition (W2) of wffs (pg 8).

Free and Bound Variables

Given the term $(\lambda x\lambda y.\, xy)y$, the result of a naive substitution as described previously (pg 8) would result in the term $\lambda y.\, yy$, which is clearly different from $\lambda y.\, xy$. The problem in this case is that the variable y was *captured* during substitution, i.e. it was substituted in a term in which it was already “bound”.

A variable x is *bound* in a term M, if M contains at least one *sub-term* of the form λxN. A sub-term can be:

- a variable
- the M in λxM
- either the N or M in MN

A variable that is not bound in a term is called *free* in that term.

For example:

The variable x is free in the term x.

Both x and y are *free* in xy.

x is bound and y is free in $\lambda x.\,xy$.

x is bound *and* free in $(xx)\lambda xx$ (bound in λxx, free in (xx)).

In general, the set of bound variables $\underline{BV}(M)$ of a term M is defined as follows.

(The notation $\underline{XY}$ is used to denote symbolic names consisting of multiple characters; the bar is used to distinguish the name “XY” from the term XY (X applied to Y).)

(BV1) $\underline{BV}(x) = \emptyset$

(BV2) $\underline{BV}(\lambda xM) = \{x\} \cup \underline{BV}(M)$

(BV3) $\underline{BV}(MN) = \underline{BV}(M) \cup \underline{BV}(N)$

For instance,

$$
\begin{aligned}
&\underline{BV}((\lambda x\lambda y.\,x)\lambda ab) \\
&= \underline{BV}(\lambda x\lambda y.\,x) \cup \underline{BV}(\lambda ab) && \text{(BV3)} \\
&= \underline{BV}(\lambda x\lambda y.\,x) \cup \{a\} \cup \underline{BV}(b) && \text{(BV2)} \\
&= \underline{BV}(\lambda x\lambda y.\,x) \cup \{a\} \cup \emptyset && \text{(BV1)} \\
&= \{x\} \cup \underline{BV}(\lambda y.\,x) \cup \{a\} && \text{(BV2)} \\
&= \{x\} \cup \{y\} \cup \underline{BV}(x) \cup \{a\} && \text{(BV2)} \\
&= \{x\} \cup \{y\} \cup \emptyset \cup \{a\} && \text{(BV1)} \\
&= \{x, y, a\}
\end{aligned}
$$

Similarly, the set $\underline{FV}(M)$ of free variables of the term M is formed as follows:

(FV1) $\underline{FV}(x) = \{x\}$

(FV2) $\underline{FV}(\lambda xM) = (\underline{FV}(M)) \setminus \{x\}$

(FV3) $\underline{FV}(MN) = \underline{FV}(M) \cup \underline{FV}(N)$

For example:

$$\begin{aligned}
&\underline{FV}(\lambda x.\, Fx) \\
&\ = \underline{FV}(Fx) \setminus \{x\} && \text{(FV2)} \\
&\ = (\underline{FV}(F) \cup \underline{FV}(x)) \setminus \{x\} && \text{(FV3)} \\
&\ = (\{F\} \cup \{x\}) \setminus \{x\} && 2 \times \text{(FV1)} \\
&\ = \{F\}
\end{aligned}$$

Programs computing the sets $\underline{BV}$ and $\underline{FV}$ can be found in the appendix (pg 149).

Substitution

Using the concept of free and bound variables, more precise rules for the *substitution* $M[x{:=}N]$ can now be defined [Church1941], see also [Hankin2004] for a more compact presentation:

(S1) $x[x{:=}N] \equiv N$

(S2) $y[x{:=}N] \equiv y$

(S3) $(\lambda xM)[x{:=}N] \equiv \lambda xM$

(S4) $(\lambda xM)[y{:=}N] \equiv \lambda x(M[y{:=}N])$,
if $x \notin \underline{FV}(N)$ or $y \notin \underline{FV}(M)$

(S5) $(\lambda xM)[y{:=}N] \equiv \lambda z(M[x{:=}z])[y{:=}N]$,
if $x \in \underline{FV}(N)$, $y \in \underline{FV}(M)$, z not in λxM

(S6) $(MN)[x{:=}P] \equiv (M[x{:=}P])(N[x{:=}P])$

The notation $M[x{:=}N]$ was introduced by Barendregt, while Church used the “substitution” operator $S^x_N M$. The notation $M[x/N]$ is also used, but less common.

Rule (S1) basically says that substituting a term N for a (free) variable x results in the term N.

Rule (S2) says that variables not subject to a specific substitution will not be affected by that substitution, e.g. substituting x will not change any occurrences of y.

Rule (S3) states that bound variables are unaffected by substitution, e.g. $(\lambda xx)[x{:=}N] \equiv \lambda xx$.

Rule (S4) defines naive substitution as described earlier (pg 8). This rules covers the case where no name capturing takes place during substitution. (Note: the $y \notin \underline{FV}(M)$ part covers the trivial case where no substitution would take place at all.)

Rule (S5) deals with name capturing. It basically says that when a name bound in a term M would be captured during substitution, that name will be replaced with a different name (z) first, where z does not appear in M (neither bound nor free). For example:

$(\lambda x \lambda y.\, xy)[x:=y]$ is converted to $(\lambda x \lambda \mathbf{z}.\, x\mathbf{z})[x:=y]$

Rule (S6), finally, covers substitution in applications by inductively applying the rules (S1) through (S6) to both the function M and argument N in (MN).

Conversion and Reduction

Conversion is the process of rewriting a term according to some specific rules. In this section, the rules of λ-conversion will be discussed, i.e. the rules for rewriting λ-terms.

Substitution rule (S5) contains the following conversion:

(Alpha) $\lambda xM \;\rightarrow_\alpha\; \lambda y(M[x:=y])$

if $y \notin \underline{FV}(M)$ and $y \notin \underline{BV}(M)$

This conversion is called *α-conversion* and it indicates that the names of bound variables can be substituted in λ-terms, as long as the new name is not already contained in the term. E.g.: λxx is the same as λyy or λaa. We say that these terms are “equal modulo α-conversion” and write:

$\lambda xx \;\equiv_a\; \lambda yy \;\equiv_a\; \lambda aa$

Note that

$\lambda xy \;\not\equiv_a\; \lambda yy$

because $y \in \underline{FV}(\lambda xy)$.

In the remainder of this text, syntactic equality will always be considered modulo α-conversion, so $M \equiv N$ will always denote $M \equiv_\alpha N$.

The application rule (App) is also known as β-conversion:

$$\text{(Beta)} \qquad (\lambda x M)N \ \to_\beta \ M[x{:=}N]$$

where $M[x{:=}N]$ indicates substitution according to the rules (S1) through (S6).

Multiple β-conversion steps may be possible in the same term:

$$\begin{aligned}
&(\lambda xyz.\, xz(yz))\ (\lambda xy.\, x)\ (\lambda xy.\, x) \\
&\qquad \to_\beta \ (\lambda yz.\, (\lambda xy.\, x)z\ (yz))\ (\lambda xy.\, x) \\
&\qquad \to_\beta \ \lambda z.\, (\lambda xy.\, x)z\ ((\lambda xy.\, x)z) \\
&\qquad \to_\beta \ \lambda z.\, (\lambda y.\, z)\ ((\lambda xy.\, x)z) \\
&\qquad \to_\beta \ \lambda z.\, (\lambda y.\, z)\ (\lambda y.\, z) \\
&\qquad \to_\beta \ \lambda zz
\end{aligned}$$

In general, M is said to be *β-reducible* (or just *reducible*) to N, if there is a finite number of reduction steps that converts M into N. This is indicated by the notation $M \twoheadrightarrow_\beta N$. Formally:

$$\text{(BR)} \qquad \frac{M \equiv N \ \text{ or } \ (M \to_\beta L \ \text{ and } \ L \twoheadrightarrow_\beta N)}{M \twoheadrightarrow_\beta N}$$

The above notation is borrowed from *sequent calculus* (see [Gentzen1934]). It is a formalization of logical reasoning, where the premises or propositions are listed above a horizontal bar and the conclusions under the bar.

When exactly one single β-reduction step is required to transform M into M, M is said to be *immediately reducible* to N. When zero or more steps are required, M is said to be *reducible* to N.

So above rule (BR) states, “if M is syntactically equal (modulo α-conversion) to N or immediately reducible to L and L is reducible to N, then M is reducible to N.”

Note that definition (BR) includes the trivial case $M \equiv N$, which requires zero reduction steps.

In the following, $M \twoheadrightarrow N$ will indicate $M \twoheadrightarrow_\beta N$ and $M \to N$ will denote $M \to_\beta N$, unless stated otherwise.

Convertibility

When either a term M is β-reducible to N or N is β-reducible to M, the terms are said to be *β-convertible* (or just convertible):

$$\text{(Conv)}\qquad \frac{M \twoheadrightarrow_\beta N \text{ or } N \twoheadrightarrow_\beta M}{M =_\beta N}$$

Note that β-reduction is reflexive ($M =_\beta M$ in zero reduction steps) and transitive (by definition) and β-convertibility is also symmetric (by definition), so convertibility is an equivalence relation:

$$M =_\beta M \qquad \text{(reflexive)}$$

$$\frac{M =_\beta N}{N =_\beta M} \qquad \text{(symmetric)}$$

$$\frac{M =_\beta L \quad L =_\beta N}{M =_\beta N} \qquad \text{(transitive)}$$

Immediate convertibility $M \rightarrow N$, on the other hand, is neither reflexive (exactly one step is required) nor transitive (if $M \rightarrow L$ and $L \rightarrow N$, then there is no single step conversion from M to N).

In the following, $M = N$ will be used in the place of $M =_\beta N$, if the meaning of the term can be understood from the context.

Redexes and Normal Forms

A term of the form $(\lambda xM)N$ is called a *β-redex*, i.e. a "β-reducible expression". (The words "term" and "*expression*" are used as synonyms here.) A term that does *not* contain any β-redexes is said to be in *β-normal form*.

In the following "redex" shall indicate a β-redex and "normal form" (or NF) shall indicate a "β-normal form".

A term M is said to "have a normal form", if $M = N$ and N is in normal form.

If a term has a NF, then that NF is unique. Two terms M and N may be considered to be equal (modulo β-reduction), if they reduce to the same NF (for a proof, see [Barendregt1985]):

Compilation

The Scheme language [R4RS] can be considered to be an implementation of λ-calculus, although there are some significant differences, as will be shown in the next chapter (pg 117).

In this chapter the interpretation and compilation of a subset of Scheme will be discussed. First, the classic model of LISP and Scheme interpretation will be introduced. Then an alternative model of interpretation will be explained and the focus will shift from λ-calculus towards Scheme. The chapter will conclude with the translation of a small Scheme subset to portable C code.

The Language

The following is a formal summary of the Scheme subset to be used in this chapter. We will call this subset $Scheme_0$ and advance the index as more definitions will be added to the language. Of course, abstraction and application are the only parts of the language that would really be required, but we add some further definitions for convenience. Definitions can be thought of to be schematic definitions (pg 31):

(L1) **(lambda (v) e)** $\rightarrow$ (λve)

(L2) **(e1 e2)** $\rightarrow$ $(e_1 e_2)$

(L3) **(if e1 e2 e3)** $\rightarrow$ $[\mathbf{if}\ e_1; e_2; e_3] \rightarrow ((e_1(\lambda xe_2)(\lambda xe_3))(\lambda xx))$

(L1), (L2), and (L3) are just alternative syntax for λ-abstraction, application, and propositional functions. Note that **lambda** is really limited to a single variable for now. This restriction will be lifted later (pg 73).

In (L3), note that e_1 will be replaced with $F \equiv (\lambda xy.\, y)$, if $e_1 \equiv \underline{nil}$ and with $T \equiv (\lambda xy.\, x)$ otherwise, thereby translating Scheme truth-values to λ-calculus.

(L4a) **(quote e)** $\rightarrow$ λxe

(L4b) **(quote e)** $\rightarrow$ $[\mathbf{nf}\ e]$

Quotation is a concept that cannot easily be translated to λ-calculus. It is used in Scheme to refer to *data objects*, a concept not known in λ-calculus. A data object may be thought of as a term that is not to be evaluated.

Hence there are two approaches to representing quoted objects. One is to use λ-abstraction to create a term in *head normal form* (L4a), the other would be to create a special schematic definition which denotes that a term already *is* in normal form (L4b).

Both approaches have their advantages and drawbacks. (L4a) maps **quote** to pure λ-calculus, but in order to access the quoted value, the resulting abstraction has to be applied to a value, which is not how quoted objects are defined in Scheme.

(L4b) avoids this problem, but at the cost of adding terms to λ-calculus that have the shapes of redexes but the semantics of normal forms.

Due to these issues quotation will be handled rather informally in the following.

The remaining definitions are merely syntactic sugar for the functions dealing with $\underline{kons}$ pairs and lists:

(L5) **(cons e1 e2)** $\rightarrow$ $(\underline{kons}\ e_1\ e_2) \equiv ((\lambda xykn.\, kxy)e_1e_2)$

(L6) **(car e)** $\rightarrow$ $(\underline{kar}\ e) \equiv ((\lambda k.\,(k\lambda xy.\, x)\lambda x.\, \underline{nil})e)$

(L7) **(cdr e)** $\rightarrow$ $(\underline{kdr}\ e) \equiv ((\lambda k.\,(k\lambda xy.\, y)\lambda x.\, \underline{nil})e)$

(L8) **(pair? e)** $\rightarrow$ $(\underline{konsp}\ e) \equiv ((\lambda k.\,(k\lambda xy.\, T)\lambda xF)e)$

(L9) **(null? e)** $\rightarrow$ $(\underline{nullp}\ e) \equiv ((\lambda k.\,(k\lambda xy.\, F)\lambda xT)e)$

Note that the propositional functions **pair?** and **null?** return **()** for falsity and **(quote t)** for truth. The **if** syntax follows [R4RS] and allows the empty list **()** to represent falsity. The more modern **#t** and **#f** forms are absent.

An Abstract Reduction Machine

Languages based on λ-calculus are typically interpreted by *abstract machines*, i.e. computer programs that implement a set of rules for interpreting formal languages.

A special case of the abstract machine is the *metacircular interpreter*, where the language being interpreted (source language, S) and the implementation language (I) of the interpreter are the same, so that primitive functions of S can directly be implemented using the corresponding functions of I. See [McCarthy1978] for an example.

In this chapter the principle of metacircular interpretation will be explored and combined with the *translation* of programs to a more suitable form for interpretation by actual computer hardware.

A Naive Compiler

Here is a simple *compiler* that translates $Scheme_0$ to a language suitable for abstract interpretation. The following transformations are performed:

Source	Target	Note
v	v	variable
(quote e)	$(\%quote\ e)$	quotation
(lambda (v) e)	$(\%lambda\ (v)\ e)$	abstraction
(if p c a)	$(\%if\ \ p\ c\ a)$	selection
(e1 e2)	$(\%apply\ e_1\ e_2)$	application
	$(\%tail\text{-}apply\ e_1\ e_2)$	application

Most terms are kept as they are, but a $\%$-prefix is added to mark keywords as machine instructions (which means that the "%" character must be excluded from the alphabet of the source language in order to avoid the accidental injection of instructions).

Only application is transformed into either an $\%apply$ or $\%tail\text{-}apply$ instruction. The type of the application instruction depends on the context in which the application appears. In each line, **a** indicates an application and **t** indicates a *tail application*:

(T1) **(if a t t)**
(T2) **(lambda (v) t)**
(T3) **(t (a ...))**

In *every* other context, application is an ordinary (non-tail) application. Note in particular: **(t (a e))**, i.e. only the outermost application in a composition of functions is in a tail position.

The meaning of tail application will be explained in a following section that deals with abstract interpretation (pg 49).

Here is the Scheme code to the compiler performing the above transformations:

```
(define (comp x)

  (define (comp-expr x t)
    (cond ((not (pair? x))
            x)
          ((eq? 'quote (car x))
            `(%quote ,(cadr x)))
          ((eq? 'lambda (car x))
            `(%lambda ,(cadr x)
                      ,(comp-expr (caddr x) #t)))
          ((eq? 'if (car x))
            `(%if ,(comp-expr (cadr x) #f)
                  ,(comp-expr (caddr x) t)
                  ,(comp-expr (cadddr x) t)))
          (else
            `(,(if t '%tail-apply '%apply)
              ,@(map (lambda (x)
                       (comp-expr x #f))
                     x)))))

  (comp-expr x #f))
```

Bindings and Free Variables

In λ-calculus, application is implemented by substitution (see rules (S1) through (S6), pg 11). However, substitution is impractical for interpretation by a computer, because it potentially has to replace variables by values in a lot of different places in a term. E.g., in

$(\lambda x.\, xxx)M$

three instances of the variable x would have to be replaced during beta reduction (rule (Beta), pg 13).

Hence an extension to λ-calculus is introduced here that allows to keep track of substitutions without actually performing them:

$(\lambda x.\, xxx)M \;\rightarrow\; [x = M]xxx$

This is an alternative syntax for the substitution

$xxx[x{:}= M]$

but we will in particular use the new notation to keep track of substitutions performed in abstractions, e.g.:

$(\lambda xy.\, x)M \;\rightarrow\; \lambda y.\, [x = M]x$

In general:

$$\begin{aligned}
&\text{(Env1)} \quad && (\lambda x_1 \cdots x_n.\, M)N_1 \cdots N_n \\
& && \equiv (\lambda x_1 \cdots x_n.\, []M)N_1 \cdots N_n \\
& && \rightarrow\; (\lambda x_2 \cdots x_n.\, [x_1 = N_1]M)N_2 \cdots N_n \\
& && \twoheadrightarrow\; [x_n = N_n; \cdots; x_1 = N_1]M
\end{aligned}$$

$$\begin{aligned}
&\text{(Env2)} \quad && [x_n = N_n; \cdots; x_1 = N_1]M \\
& && \equiv\; M[x_n = N_n] \cdots [x_1 = N_1] \\
& && \text{if } x_1, \cdots x_n \notin \bigcup_{i=1}^{n} \underline{FV}(N_i)
\end{aligned}$$

The $[x = M]$ part in $\lambda y.\, [x = M]N$ is called the *environment* of the term. An environment consists of a list of *variable-term* pairs $x = M$, where each pair indicates that the variable x is *bound to* the value M. Each variable in the term N that appears in the environment is to be replaced with the value to which it is bound. An environment may be empty, which is denoted by [].

This approach is equivalent to multiple (i.e. zero or more) substitutions as shown in (Env2), as long as no variable in the environment occurs free in a term of the environment. We will later see that this precondition is trivially fulfilled in the implementation (pg 54).

Note particularly that, due to (Env1),

$$\begin{aligned}(\lambda xx.\,M)NP &\rightarrow (\lambda x.\,[x=N]M)P\\ &\rightarrow [x=P;\,x=N]M\\ &\equiv M[x=P]\end{aligned}$$

because after the substitution $M[x=P]$, there are no free instances of x left in M. Technically speaking, variables of inner scopes take precedence over variables of outer scopes. This phenomenon is known as *shadowing* of variables in programming jargon. It is consistent with the reduction

$$(\lambda xx.\,x)NP \twoheadrightarrow P$$

Abstract Interpretation

The code presented in this section *evaluates* expressions compiled by the compiler discussed previously. In this context, “evaluating an expression” basically means to reduce it to its normal form mechanically. All evaluators described in the following reduce *restricted* $\lambda K\beta$-calculus and do not reduce head normal forms.

The abstract machine performing the reduction operates on four objects:

- the expression x
- the environment E
- the stack S
- the continuation C

The *expression* x is a compiled $Scheme_0$ program as described in the previous sections.

The *environment* E is an ordered set of variable bindings as described in the previous section, like the $[x=N]$ part in the term $\lambda y.\,[x=N]M$. Note that E is merely a global pointer to the (possibly local) environment currently in effect. More on that below.

List of Figures

List of Definitions

Index