Notes on evaluating λ-calculus terms and abstract machines

J.R.B. Cockett
Department of Computer Science, University of Calgary
Calgary, T2N 1N4, Alberta, Canada

November 17, 2016

1 Introduction

In this document we discuss evaluation strategies for λ-terms. We shall present six different evaluation strategies. They are distinguished by whether they are by-value (basically innermost) or by-name (basically outermost) and what sort of normal form they aim to produce. There are three normal forms weak head normal form, head normal form, and normal form. To implement the λ-calculus one must take the further step of translating these reduction strategies into the actions of an abstract machine for which one compiles λ-terms into code. This makes for very simple yet powerful evaluation techniques which are relatively efficient. We illustrate this with two machines: a "modern" version of Landin's SECD machine and the Krivine machine.

2 Evaluation strategies

There are two basic families of evaluation strategies (reduction strategies) for the λ-calculus: "by-value" and "by-name". The former family of evaluation strategy, "by-value", evaluates arguments (and sometimes function bodies) before applying a function to its arguments. The "by-value" strategies are relatively simple to implement and are reasonably efficient, however, they suffers from a significant defect: they may not find a normal form (or a weak head normal form) even if there is one. The "by-name" evaluation strategy, are (perhaps) more complex to implement as they leave the arguments unevaluated: this can make the evaluation very inefficient as unevaluated terms can be duplicated thereby doubling the cost of their evaluation. However, the "by-name" strategies have an important theoretical advantage: if there is a normal form (weak head normal form) they will find it. Because of this considerable effort has been into to making these strategies more efficient and this has led to the development of "graph reduction" techniques for "lazy evaluation": these underly the evaluation of Haskell programs for example. In graph reduction the duplicated terms are shared and, thus, are only evaluated once: in terms of the number of β-reduction steps, lazy evaluation does the minimal number possible: however, there is an overhead to sharing terms – in order to keep track of when a term has been already evaluated – which will sometimes make it less efficient than the simpler by-value evaluation.

Both "by-value" and "by-name" strategies come in different forms depending on where the evaluation terminates – if it does terminate. The "weak" strategies aim to produce a result in weak head normal form. The essential feature of the weak evaluation strategies is that they *never*

evaluate the bodies of λ-abstractions. These evaluation strategies are closely related to what is actually implemented in functional languages: the tendency is to want to compile (and optimize) function bodies and thus to never modify them while the program is running. The "strong" evaluation strategies aim to produce a term in η-normal form. Strong reduction strategies are primarily of theoretical interest: for example "strong by-name" evaluation is a leftmost outermost reduction – often called "normal order reduction" – and is guaranteed to find a normal form if there is one. Finally there are the "head" reduction strategies: these evaluate the term to head normal form. Head reduction strategies do go inside the top level λ-abstractions but no others. Head evaluation is also closely related to evaluation strategies for functional languages: when constructors for datatypes are not built in and are natively expressed in the λ-calculus they involve a top level binding which the evaluation must go under.

Lastly a remark: writing down formally how an evaluation strategy works may seem easy ..... but, in fact, it is not so easy! The crucial issue is: how exactly does one efficiently find the next redex to reduce? For example, saying that one always reduces the "leftmost outermost" redex tells one how to find the next redex from the global perspective of the whole term: however, an implementation of an evaluation which searches for redexes by starting from scratch at each step would be very inefficient and this is never done in practice. Thus, more detailed descriptions are necessary. To this end, below, we use both inference systems and recursive descriptions of the evaluation techniques. (One reason for using inference systems is actually because you must get used to using them!) These lead ultimately to the abstract machines which *are* used in practice for evaluation.

2.1 By-value evaluation strategies

We shall start by discussing "by-value" evaluation strategies. We shall discuss two of these by-value strategies. The first first strategy is the"innermost" strategy: here whenever a reduction is performed *all* the subexpressions of the redex must be in normal form. Thus the reduction

$$C[\![(\lambda x.M)\ N]\!] \underset{\beta}{\longrightarrow} C[\![M[N/x]]\!]$$

can only be performed once M and N have been reduced to normal form. The second strategy is the "rightmost" evaluation strategy here the above reduction can be performed but only when N is in normal form. This means that in rightmost evaluation one does not evaluate function bodies - λ-abstractions before applying them. This from a programming perspective is very natural as normally one wishes to compile function bodies into code which one does not rewrite.

2.1.1 Innermost evaluation

This is a strong reduction strategy. As described above, in this strategy one can only perform a β-reduction when the subexpressions of the redex are already in normal form. This means that when one encounters a redex one must recursively reduce all it subexpressions – this can be done in parallel – before one performs the reduction of that redex. This does have the slightly peculiar property that one reduces the body of functions before one applies them and this is not usually how functional languages are implemented.

The strategy can be described by the inference system in Table 1. The inferences in the system are to be read as saying that, in order to conclude the statement below the line, one must first have

$$\frac{\begin{array}{c} M \rightsquigarrow \lambda x.M' \\ N \rightsquigarrow N' \\ (\lambda x.M')\ N' \xrightarrow{\beta} L' \\ L' \rightsquigarrow L \end{array}}{M\ N \rightsquigarrow L}\ \text{Evaluate to abstraction}$$

$$\frac{\begin{array}{c} M \rightsquigarrow P\ Q \\ N \rightsquigarrow N' \end{array}}{M\ N \rightsquigarrow (P\ Q)\ N'}\ \text{Evaluate to application}$$

$$\frac{\begin{array}{c} M \rightsquigarrow y \\ N \rightsquigarrow N' \end{array}}{M\ N \rightsquigarrow y\ N'}\ \text{Evaluate to variable}$$

$$\frac{\mathsf{Var}(x)}{x \rightsquigarrow x}\ \text{Variable} \qquad \frac{M \rightsquigarrow M'}{\lambda x.M \rightsquigarrow \lambda x.M'}\ \text{Abstraction}$$

Table 1: Innermost Evaluation

concluded all the things above the line. Thus, the first rule of Table 1 says that, in order evaluate $M\ N$ to L – written as $M\ N \rightsquigarrow L$ below the line – one must:

(a) Evaluate M to an abstraction of the form (here we are working up to α-equality) $\lambda x.M'$. This is indicated by

$$M \rightsquigarrow \lambda x.M'$$

If, in fact, M *does not* evaluate to an abstraction we will use one of the next two rules which are designed to cover the other possible cases.

(b) Evaluate N to N', written $N \rightsquigarrow N'$.

(c) Perform the β-reduction $(\lambda x.M')\ N' \xrightarrow{\beta} L'$.

(d) Evaluate L' to L, that is $L \rightsquigarrow L'$.

Consider the following examples:

Example 2.1

(1) Consider the by-value evaluation of $(\lambda x.(\lambda y.yy)x)z$:

$$\dfrac{\dfrac{\dfrac{\vdots}{(a)\ \overline{\lambda y.y\ y \rightsquigarrow \lambda y.y\ y}}\quad \dfrac{\mathsf{var}(x)}{(b)\ \overline{x \rightsquigarrow x}}}{(c)\ (\lambda y.y\ y)\ x \underset{\beta}{\longrightarrow} x\ x}\quad \dfrac{\dfrac{\vdots}{(d)\ \overline{x\ x \rightsquigarrow x\ x}}}{(\lambda y.y\ y)\ x \rightsquigarrow x\ x}}{(a)\ (\lambda x.(\lambda y.y\ y)\ x) \rightsquigarrow \lambda x.x\ x}\quad (b)\ \overline{z \rightsquigarrow z}$$
$$(c)\ (\lambda x.x\ x)\ z \underset{\beta}{\longrightarrow} z\ z$$

$$\dfrac{\dfrac{\vdots}{(d)\ \overline{z\ z \rightsquigarrow z\ z}}}{(\lambda x.(\lambda y.y\ y)\ x)\ z \rightsquigarrow z\ z}$$

(2) Notice that $(\lambda xy.y)\ \Omega$ does not have a terminating innermost by-value reduction as one must evaluate $\Omega := (\lambda x.x\ x)\ (\lambda x.x\ x)$ before one can reduce β-reduction at the root of this term: the reduction of Ω, of course, never terminates. Similarly, $(\lambda xy.y)\ (\lambda y.\Omega)$ will not have a terminating innermost reduction because of the presence of Ω.

(3) We shall use the evaluation of $\mathsf{square}\ \mathsf{square}\ 2$ as a running example where

$$\mathsf{square} := \lambda x.x * x$$

In addition we shall, to facilitate this example, allow ourselves the assumption that when two numbers are multiplied they can be reduced to the answer, e.g. $3*4 \rightsquigarrow 12$. Here is the structure of the by-value evaluation:

$$\dfrac{\dfrac{\dfrac{\vdots}{(a)\ \overline{\mathsf{square} \rightsquigarrow \mathsf{square}}}\quad \dfrac{\dfrac{\vdots}{(a)\ \overline{\mathsf{square} \rightsquigarrow \mathsf{square}}}\quad \dfrac{\vdots}{(b)\ \overline{2 \rightsquigarrow 2}}}{(c)\ \mathsf{square}\ 2 \underset{\beta}{\longrightarrow} 2 * 2}\quad \dfrac{\vdots}{(d)\ \overline{2 * 2 \rightsquigarrow 4}}}{(b)\quad \mathsf{square}\ 2 \rightsquigarrow 4}}{(c)\ \mathsf{square}\ 4 \underset{\beta}{\longrightarrow} 4 * 4}\quad \dfrac{\dfrac{\vdots}{(d)\ \overline{4 * 4 \rightsquigarrow 16}}}{\mathsf{square}\ (\mathsf{square}\ 2) \rightsquigarrow 16}$$

Notice how every subexpression gets evaluated before the whole expression. Also how one normalizes the function bodies: here they are already taken as being in normal form.

We may also express the innermost reduction strategy very simply as a recursive function on λ-terms:

$$\begin{array}{rcl} \mathcal{I}(x) & = & x \\ \mathcal{I}(x\ N) & = & x\ \mathcal{I}(N) \\ \mathcal{I}((\lambda x.M)\ N) & = & \mathcal{I}(\mathcal{I}(M)[\mathcal{I}(N)/x]) \\ \mathcal{I}((M\ N)\ P) & = & \begin{cases} L[\mathcal{I}(N)/y] & \mathcal{I}(M\ N) = \lambda y.L \\ \mathcal{I}(M\ N)\ \mathcal{I}(M) & \text{otherwise} \end{cases} \end{array}$$

2.1.2 Rightmost evaluation

The basic reduction strategy adopted by SML and OCaml is a "weak" rightmost reduction strategy. This means arguments to functions are evaluated before functions are applied to them and it is "weak" because functions are never evaluated. The "strong" rightmost by-value reduction strategy, however, does allow function bodies to be evaluated but only when they are not applied to any argument. This makes it more complicated to describe as the reduction involves aspects of weak reduction until it is discovered that the λ-term cannot ever be applied when the reduction is "pushed" under the λ-abstractions.

2.1.3 Weak rightmost evaluation

We shall start by describing the weak rightmost reduction system. The "weak" reduction strategies reduce λ-terms to what is called **weak head normal form**. Thus,

$$t \rightsquigarrow_w \mathsf{whnf}(t)$$

when the reduction terminates – we shall see shortly that Ω has no weak head normal form. By way of contrast, the "strong" rightmost reduction strategy, if it terminates, will do so only at a normal form, thus $t \rightsquigarrow_s \mathsf{nf}(t)$.

A λ-term, M, is in **weak head normal form**[1] in case it is a λ-abstraction, $M = \lambda x.N$, or it is of the form $M = x\ N_1\ ...\ N_p$ where x is any variable. Shortly we shall see that why it is called a *weak* head normal form it is a head normal form which, in addition could be a λ-abstraction.

The weak rightmost reduction strategy is described in Table 2. The key difference is that when a λ-abstraction is encountered the evaluation "stops" and never goes inside/under the abstraction. Notice also that, the system does evaluate arguments in a rightmost fashion, although the evaluation which distinguishes the cases has been put first for clarity. In fact, the order of evaluation of the arguments does not actually matter and, indeed, can be done in parallel. Thus, if we take the first rule what it says in detail is:

In order to evaluate $M\ N$ to L:,

(a) Evaluate N to N'.

[1] Weak head normal forms were introduced by Simon Peyton Jones to reflect the form to which functional languages actually evaluate.

$$\frac{(b)\ M \rightsquigarrow_w \lambda x.M' \quad (a)\ N \rightsquigarrow_w N' \quad (c)\ (\lambda x.M')\ N' \xrightarrow{\beta} L' \quad (d)\ L' \rightsquigarrow_w L}{M\ N \rightsquigarrow_w L}\ M\text{-evaluate to abstraction}$$

$$\frac{(b)\ M \rightsquigarrow_w P\ Q \quad (a)\ N \rightsquigarrow_w N'}{M\ N \rightsquigarrow_w (P\ Q)\ N'}\ M \text{ evaluate to application}$$

$$\frac{(b)\ M \rightsquigarrow_w y \quad (a)\ N \rightsquigarrow_w N'}{M\ N \rightsquigarrow_w y\ N'}\ M \text{ evaluate to variable}$$

$$\frac{\mathsf{Var}(x)}{x \rightsquigarrow_w x}\ \text{Variable} \qquad \frac{\lambda\text{-abstraction}}{\lambda x.M \rightsquigarrow \lambda x.M}\ \text{Weak reduction}$$

Table 2: Weak Rightmost Evaluation

(b) Evaluate M to an abstraction $\lambda x.M'$. If it does not evaluate to an abstraction use the appropriate one of the rules which follow ... note that each requires that (a) be performed.

(c) Perform the β-reduction $(\lambda x.M')N' \underset{\beta}{\longrightarrow} L'$.

(d) Evaluate L' to L.

The system even if one evaluates the arguments in a different order, does ensure that arguments are evaluated before functions are applied: this is the key feature of by-value evaluation.

Example 2.2

(1) Consider the weak rightmost evaluation of $(\lambda xy.y)\ (\lambda y.y\ \Omega)$

$$\frac{(a)\ \dfrac{\lambda\text{-abstraction}}{(\lambda y.y\ \Omega) \rightsquigarrow_w (\lambda y.y\ \Omega)} \quad (b)\ \dfrac{\lambda\text{-abstraction}}{(\lambda xy.y) \rightsquigarrow_w (\lambda xy.y)} \quad (c)\ (\lambda xy.y)\ (\lambda y.y\ \Omega) \underset{\beta}{\longrightarrow} \lambda y.y \quad (d)\ \dfrac{\lambda\text{-abstraction}}{(\lambda y.y) \rightsquigarrow_w (\lambda y.y)}}{(\lambda xy.y)\ (\lambda y.y\ \Omega) \rightsquigarrow_w \lambda y.y}$$

This shows that under weak rightmost reduction (here we have swapped (a) and (b) to get strict rightmost reduction) one can have non-terminating subterms which never get evaluated. Note that the innermost reduction strategy will not terminate on this term.

(2) Note that weak rightmost reduction does not terminate on Ω nor does it terminate on $(\lambda xy.y)\ \Omega$.

We may also express the weak rightmost reduction using a recursive algorithm as follows:

$$\begin{array}{rcl} \mathcal{R}_w(\lambda x.L) & = & \lambda x.L \\ \mathcal{R}_w(x) & = & x \\ \mathcal{R}_w(N\ M) & = & \begin{cases} \mathcal{R}_w(M'[\mathcal{R}_w(N)/y] & \mathcal{R}_w(M) = \lambda y.M' \\ \mathcal{R}_w(M)\ \mathcal{R}_w(N) & \text{otherwise} \end{cases} \end{array}$$

2.1.4 Strong rightmost evaluation

The idea of strong rightmost evaluation is to perform a rightmost evaluation which *does* end at a normal form when it terminates. This means, in particular, that the evaluation will sometimes have to go inside λ-abstractions but only when it is not the argument of a β-reduction. The reduction strategy is described in Table 3: notice that is more complex as one must flip from a weak reduction strategy to a strong reduction when the head term is not a λ-abstraction and it is necessary to force the evaluation of the arguments. This does mean that one needs to know how the head term evaluates before deciding whether to initiate a strong reduction of the argument. This system is largely a curiosity! It illustrates the technique of getting a full reduction by having a mutually recursive reduction strategy.

It is perhaps worth mentioning that this reduction system is best implemented as first evaluating the function to determine how the argument needs to be evaluated (either strongly or weakly).

This may be expressed as recursive function by:

$$\begin{array}{rcl} \mathcal{R}_s(x) & = & x \\ \mathcal{R}_s(\lambda x.M) & = & \lambda x.\mathcal{R}_s(M) \\ \mathcal{R}_s(M\ N) & = & \begin{cases} \mathcal{R}_s(M'[\mathcal{R}_{ws}(N)/y]) & \mathcal{R}_{ws}(M) = \lambda y.M' \\ \mathcal{R}_{ws}(M)\ \mathcal{R}_s(N) & \text{otherwise} \end{cases} \\ \mathcal{R}_{ws}(x) & = & x \\ \mathcal{R}_{ws}(\lambda x.M) & = & \lambda x.M \\ \mathcal{R}_{ws}(M\ N) & = & \begin{cases} \mathcal{R}_{ws}(M'[\mathcal{R}_{ws}(N)/y]) & \mathcal{R}_{ws}(M) = \lambda y.M' \\ \mathcal{R}_{ws}(M)\ \mathcal{R}_s(N) & \text{otherwise} \end{cases} \end{array}$$

2.2 By-name evaluation strategies

These are basically "outermost" evaluation strategies. In an outermost evaluation strategy one always performs β-reductions steps which are closer to the root of the term *before* those which are further from the root. This therefore is exactly the opposite of an innermost evaluation strategy. Thus, if two redexes are nested on must always choose to develop the one closest to the root first. Of course, this does not completely determine the order so one usually combines this with a "leftmost" strategy which requires that the one always first evaluates the leftmost redex of any two redexes which are in parallel, in the sense of being on different branches of the tree: this is called the "leftmost outermost" evaluation strategy (were on reads "leftmost" as a modifier on the basic "outermost" strategy).

$$\frac{(b)\ M \rightsquigarrow_{ws} \lambda x.M' \quad (a)\ N \rightsquigarrow_{ws} N' \quad (c)\ (\lambda x.M')\ N' \xrightarrow{\beta} L' \quad (d)\ L' \rightsquigarrow_{s} L}{M\ N \rightsquigarrow_{s} L}\ M\text{-evaluate to abstraction}$$

$$\frac{(b)\ M \rightsquigarrow_{ws} P\ Q \quad (a)\ N \rightsquigarrow_{s} N'}{M\ N \rightsquigarrow_{s} (P\ Q)\ N'}\ M \text{ evaluate to application}$$

$$\frac{(b)\ M \rightsquigarrow_{ws} y \quad (a)\ N \rightsquigarrow_{s} N'}{M\ N \rightsquigarrow_{s} y\ N'}\ M \text{ evaluate to variable}$$

$$\frac{\mathsf{Var}(x)}{x \rightsquigarrow_{s} x}\ \text{Variable} \qquad \frac{M \rightsquigarrow_{s} M'}{\lambda x.M \rightsquigarrow_{s} \lambda x.M'}\ \text{Strong reduction}$$

$$\frac{(b)\ M \rightsquigarrow_{ws} \lambda x.M' \quad (a)\ N \rightsquigarrow_{ws} N' \quad (c)\ (\lambda x.M')\ N' \xrightarrow{\beta} L' \quad (d)\ L' \rightsquigarrow_{ws} L}{M\ N \rightsquigarrow_{ws} L}\ M\text{-evaluate to abstraction}$$

$$\frac{(b)\ M \rightsquigarrow_{ws} P\ Q \quad (a)\ N \rightsquigarrow_{s} N'}{M\ N \rightsquigarrow_{ws} (P\ Q)\ N'}\ M \text{ evaluate to application}$$

$$\frac{(b)\ M \rightsquigarrow_{ws} y \quad (a)\ N \rightsquigarrow_{s} N'}{M\ N \rightsquigarrow_{ws} y\ N'}\ M \text{ evaluate to variable}$$

$$\frac{\mathsf{Var}(x)}{x \rightsquigarrow_{ws} x}\ \text{Variable} \qquad \frac{\lambda\text{-abstraction}}{\lambda x.M \rightsquigarrow_{ws} \lambda x.M}\ \text{Weak reduction}$$

Table 3: Strong Rightmost Evaluation

$$\frac{(a)\ M \rightsquigarrow_w \lambda x.M' \quad (b)\ (\lambda x.M')\ N \xrightarrow{\beta} L' \quad (c)\ L' \rightsquigarrow_w L}{M\ N \rightsquigarrow_w L}\ \text{Leftmost (to abstraction)}$$

$$\frac{(a)\ M \rightsquigarrow_w P\ Q \quad (b)\ N \rightsquigarrow_w N'}{M\ N \rightsquigarrow_w (P\ Q)\ N'}\ \text{Leftmost (to application)}$$

$$\frac{(a)\ M \rightsquigarrow_w y \quad (b)\ N \rightsquigarrow_w N'}{M\ N \rightsquigarrow_w y\ N'}\ \text{Leftmost (to variable)}$$

$$\frac{\mathsf{Var}(x)}{x \rightsquigarrow_w x}\ \text{Variable} \qquad \frac{\lambda\text{-abstraction}}{\lambda x.M \rightsquigarrow_w \lambda x.M}\ \text{Weak reduction}$$

Table 4: Weak by-name evaluation

2.2.1 Weak by-name evaluation

This is essentially the strategy underlying lazy functional languages except, of course, that these languages use graph reduction techniques to avoid the duplications of subterms which is caused by a β-reduction. The reduction here is to weak head normal form. Thus, the evaluation never goes inside/under a λ-abstraction. The strategy is described in Table 4 and, if it terminates will do so on a weak head normal form. The strategy has the merit of being remarkably simple.

As mentioned a defect of by-name evaluation is that it can duplicate unevaluated terms, this duplicate the evaluation work making the technique very inefficient. Recall that this problem can be overcome by using graph reduction techniques to share the duplicated terms. Below is an simple illustration of the problem:

Example 2.3

(1) Consider the evaluation of $(\lambda xy.y)\ \Omega\ (\lambda z.N)$:

$$\dfrac{\begin{array}{ll}(a) & \dfrac{\lambda\text{-abstraction}}{\lambda xy.y \rightsquigarrow \lambda xy.y} \\ (b) & (\lambda xy.y)\ \Omega \underset{\beta}{\longrightarrow} \lambda y.y \\ (c) & \dfrac{\lambda\text{-abstraction}}{\lambda y.y \rightsquigarrow \lambda y.y}\end{array}}{\begin{array}{ll}(a) & (\lambda xy.y)\ \Omega \rightsquigarrow \lambda y.y \\ (b) & (\lambda y.y)\ (\lambda z.N) \underset{\beta}{\longrightarrow} (\lambda z.N) \\ (c) & \dfrac{\lambda\text{-abstraction}}{\lambda z.N \rightsquigarrow \lambda z.N}\end{array}}$$
$$(\lambda xy.y)\ \Omega\ (\lambda z.N) \rightsquigarrow \lambda z.N$$

Notice that this evaluates a weak head normal form: Ω, in contrast to the by-value strategies, is never evaluated and, because N is the body of an abstraction we never have to reduce this.

(2) Consider the evaluation of square square 2 where

$$\mathsf{square} := \lambda x.x * x$$

Again, to facilitate this example, allow ourselves the assumption that when two numbers are multiplied they can be reduced to the answer, e.g. $4 * 4 \rightsquigarrow 12$ and that one must to evaluate two multiplied expressions one must minimal evaluate each expression. Here is then (roughly) the structure of the weak by-name evaluation of this expression:

$$\dfrac{\begin{array}{ll}(a) & \dfrac{\text{defn. and } \lambda\text{-abstraction}}{\mathsf{square} \rightsquigarrow \lambda x.x * x} \\ (b) & (\lambda x.x * x)\ (\mathsf{square}\ 2) \underset{\beta}{\longrightarrow} (\mathsf{square}\ 2) * (\mathsf{square}\ 2) \\ (c) & \dfrac{\dfrac{\vdots}{\mathsf{square}\ 2 \rightsquigarrow 4} \quad \dfrac{\dfrac{\vdots}{\mathsf{square2} \rightsquigarrow 4} \quad 4 * 4 \longrightarrow 16}{4 * (\mathsf{square2}) \rightsquigarrow 16}}{(\mathsf{square}\ 2) * (\mathsf{square}\ 2) \rightsquigarrow 16}\end{array}}{\mathsf{square}\ (\mathsf{square}\ 2) \rightsquigarrow 16}$$

Notice how subexpression are not evaluated but duplicated leading to duplicated the evaluation work.

We may write this evaluation strategy as a recursive function:

$$\begin{array}{rcl}\mathcal{N}_w(x) & = & x \\ \mathcal{N}_w(\lambda x.M) & = & \lambda x.M \\ \mathcal{N}_w(M\ N) & = & \begin{cases}\mathcal{N}_w(M'[N/y]) & \mathcal{N}_w(M) = \lambda y.M' \\ \mathcal{N}_w(M)\ \mathcal{N}_w(N) & \text{otherwise}\end{cases}\end{array}$$

2.2.2 Strong by-name evaluation

As before the purpose of strong by-name reduction is to obtain a normal form rather than a weak head normal form. This means the reduction strategy must sometimes be forced to go inside λ-abstractions: controlling this aspect of the evaluation causes the strategy to be more complex. As above this leads to needing a second evaluation steps $N \rightsquigarrow_{ws} N'$ which recursively used the strong evaluation $N \rightsquigarrow_{s} N'$. The evaluation strategy is described in Table 5 together with its mutually recursive definition.

The importance of this evaluation strategy is that, if there is a normal form for N, then this reduction strategy will terminate by finding that normal form. This is proven using the "standardization theorem" which says that any reduction sequence in the λ-calculus can be rearranged to be an outermost reduction and this in turn can be rearranged to be a leftmost outermost reduction.

2.3 Head evaluation

The last evaluation strategy is to **head normal form**: a λ-term is in head normal form if it is of the form

$$\lambda x_1 ... x_n . y \ N_1 ... N_p$$

where $n, p \in \mathbb{N}$ and so include the possibility of being 0! To evaluate to head normal form it is necessary to evaluate inside the topmost λ-abstractions. This means again that the expression of the reduction strategy is more complex and requires a mutual recursion. This is called a "head' reduction" because , after going inside top-level λ-abstractions, one searches for reductions on the application chains by repeatedly looking down the left branch until a non-application is found: this is the head term of the application chain. If the head term is a λ-abstraction one has a reduction. One simply repeats this process until the head is a variable.

$$\frac{(a)\ M \rightsquigarrow_w \lambda x.M' \quad (b)\ (\lambda x.M')\ N \xrightarrow{\beta} L' \quad (c)\ L' \rightsquigarrow L}{M\ N \rightsquigarrow L}\ \text{Leftmost (to abstraction)}$$

$$\frac{(a)\ M \rightsquigarrow_w P\ Q \quad (b)\ N \rightsquigarrow N'}{M\ N \rightsquigarrow (P\ Q)\ N'}\ \text{Leftmost (to application)}$$

$$\frac{(a)\ M \rightsquigarrow_w y \quad (b)\ N \rightsquigarrow N'}{M\ N \rightsquigarrow y\ N'}\ \text{Leftmost (to variable)}$$

$$\frac{\mathsf{Var}(x)}{x \rightsquigarrow x}\ \text{Variable} \qquad \frac{M \rightsquigarrow M'}{\lambda x.M \rightsquigarrow \lambda x.M'}\ \text{Abstraction}$$

$$\frac{(a)\ M \rightsquigarrow_w \lambda x.M' \quad (b)\ (\lambda x.M')\ N \xrightarrow{\beta} L' \quad (c)\ L' \rightsquigarrow_w L}{M\ N \rightsquigarrow_w L}\ \text{Leftmost (to abstraction)}$$

$$\frac{(a)\ M \rightsquigarrow_w P\ Q \quad (b)\ N \rightsquigarrow N'}{M\ N \rightsquigarrow_w (P\ Q)\ N'}\ \text{Leftmost (to application)}$$

$$\frac{(a)\ M \rightsquigarrow_w y \quad (b)\ N \rightsquigarrow N'}{M\ N \rightsquigarrow_w y\ N'}\ \text{Leftmost (to variable)}$$

$$\frac{\mathsf{Var}(x)}{x \rightsquigarrow_w x}\ \text{Variable} \qquad \frac{\text{Abstraction}}{\lambda x.M \rightsquigarrow_w \lambda x.M}\ \text{Weak reduction}$$

$$\begin{array}{rcl}
\mathcal{N}(x) & = & x \\
\mathcal{N}(\lambda x.M) & = & \lambda x.\mathcal{N}(M) \\
\mathcal{N}(M\ N) & = & \begin{cases} \mathcal{N}(M'[N/y]) & \mathcal{N}_w(M) = \lambda y.M' \\ \mathcal{N}_w(M)\ \mathcal{N}(N) & \text{otherwise} \end{cases} \\
\mathcal{N}_w(x) & = & x \\
\mathcal{N}_w(\lambda x.M) & = & \lambda x.M \\
\mathcal{N}_w(M\ N) & = & \begin{cases} \mathcal{N}_w(M'[N/y]) & \mathcal{N}_w(M) = \lambda y.M' \\ \mathcal{N}_w(M)\ \mathcal{N}(N) & \text{otherwise} \end{cases}
\end{array}$$

Table 5: Strong by-name evaluation – normal order reduction

$$\frac{(a)\ M \rightsquigarrow_w \lambda x.M' \quad (b)\ (\lambda x.M')\ N \xrightarrow{\beta} L' \quad (c)\ L' \rightsquigarrow L}{M\ N \rightsquigarrow L}\ \text{Head evaluate (to abstraction)}$$

$$\frac{M \rightsquigarrow_w P\ Q \quad N \rightsquigarrow N'}{M\ N \rightsquigarrow (P\ Q)\ N}\ \text{Head evaluate (to application)}$$

$$\frac{M \rightsquigarrow_w y}{M\ N \rightsquigarrow y\ N}\ \text{Head evaluate (to variable)}$$

$$\frac{\mathsf{Var}(x)}{x \rightsquigarrow x}\ \text{Variable} \qquad \frac{M \rightsquigarrow M'}{\lambda x.M \rightsquigarrow \lambda x.M'}\ \text{Abstraction}$$

$$\frac{(a)\ M \rightsquigarrow_w \lambda x.M' \quad (b)\ (\lambda x.M')\ N \xrightarrow{\beta} L' \quad (c)\ L' \rightsquigarrow_w L}{M\ N \rightsquigarrow_w L}\ \text{Head evaluate (to abstraction)}$$

$$\frac{M \rightsquigarrow_w P\ Q \quad N \rightsquigarrow N'}{M\ N \rightsquigarrow_w (P\ Q)\ N}\ \text{Head evaluate (to application)}$$

$$\frac{M \rightsquigarrow_w y}{M\ N \rightsquigarrow_w y\ N}\ \text{Head evaluate (to variable)}$$

$$\frac{\mathsf{Var}(x)}{x \rightsquigarrow_w x}\ \text{Variable} \qquad \frac{\text{Abstraction}}{\lambda x.M \rightsquigarrow_w \lambda x.M}\ \text{Weak reduction}$$

$$\begin{array}{rcl}
\mathcal{H}(x) & = & x \\
\mathcal{H}(\lambda x.M) & = & \lambda x.\mathcal{H}(M) \\
\mathcal{H}(M\ N) & = & \begin{cases} \mathcal{H}(M'[N/y]) & \mathcal{H}_w(M) = \lambda y.M' \\ \mathcal{H}_w(M)\ \mathcal{H}(N) & \text{otherwise} \end{cases} \\
\mathcal{H}_w(x) & = & x \\
\mathcal{H}_w(\lambda x.M) & = & \lambda x.M \\
\mathcal{H}_w(M\ N) & = & \begin{cases} \mathcal{H}_w(M'[N/y]) & \mathcal{H}_w(M) = \lambda y.M' \\ \mathcal{H}_w(M)\ \mathcal{H}(N) & \text{otherwise} \end{cases}
\end{array}$$

Table 6: Head evaluation

3 Abstract machines

Abstract machines reorganize the evaluation into simple machine transitions which one aims to make nearly constant time steps. One then starts the machine in a state determined by the desired computation and repeatedly do transitions until a final state is reached when one publishes the answer! If one arranges each step to correspond to an instruction then one can compile λ-terms down to a sequence of instructions. Reorganizing the computation into this from allows one to compile λ-terms into code – sequences of instructions – and introduces the opportunity for more efficient and simpler evaluation due to this compilation step.

We shall describe two abstract machines below: a modern version of Landin's SECD machine and the Krivine machine.

3.1 The CES machine or the modern SECD machine

The modern SECD machine, which is the CES machine, is a simplification of Landin's original SECD machine (see Paulson's notes). It implements a weak by-value reduction of the λ-term by compiling the term into CES code which is run to obtain the weak head normal form (if it exists) of the λ-term. There are two improvements over Landin's SECD machine: it does not have a dump – as the stack doubles up as a dump – and it uses a simpler instruction set.

The CES machine is a state machine with transitions: the state is a triple consisting of:

C: A **code** pointer, C, which points to the current instruction.

E: An **environment**, E, which holds the values of variables. These are accessed using De-Bruijn's Indices.

S: A **stack** S holding both intermediate results continuations (closures).

Here we shall illustrate the CES machine with additional instructions for integers, Booleans, and Lists. This means that we must augment the λ-calculus with arithmetic instructions and list instructions. For arithmetic we add "constants" (meaning integers, k), with instructions for addition, Add, multiplication, Mul, and comparison leq. For booleans we add constants True and False and the conditional If. For lists we add the constants Nil and Cons and the case instruction.

The resulting instructions for a CES machine are:

Instruction	Explanation
Clo(c)	Push closure of code c with current environment on the stack
App	Pop function closure and argument, perform application
Access(n)	Push n^{th} value in the environment onto the stack.
Ret	Return the top value on the stack and jump to the continuation on the stack
Arithmetic instructions:	
Const(n)	push the constant n on the stack
Add	Pop two arguments from the top of the stack and add them
Mul	Pop two arguments from the top of the stack and add them
Leq	Pop two arguments from the top of the stack and compare them
Boolean instructions:	
True	Push the constant True on the stack
False	Push the constant False on the stack
If	Pop an argument from the top of the stack and depending on whether it is True or False evaluate the appropriate branch.
List instructions:	
Cons	push the Cons applied to the top two elements of the stack onto the stack
Nil	push the constant Nil on the stack
Case(c_1, c_2)	if Cons t_1 t_2 is on the stack pop it and push t_2 and t_1 onto the environment. Evaluate c_2
Case(c_1, c_2)	if Nil pop it and evaluate c_1

To compile a λ-term into CES code one essentially converts the λ-term into de Bruijn notation and then translates the term into CES-machine code. This one can do in two steps. However, below we present it as one big step. Recall the translation into de Bruijn notation replaces variables by indexes:

Example 3.1 Here is the De Bruijn notation for the following λ-terms:

$$\begin{aligned} (\lambda x.xx)\ (\lambda x.x) &\Rightarrow (\lambda.(\#1\ \#1))(\lambda.\ \#1) \\ (\lambda x\ .\lambda y\ .(xy))\ (\lambda x\ .x)\ (\lambda y\ .y) &\Rightarrow (\lambda.\ \lambda.(\#2\ \#1))(\lambda.\#1)(\lambda.\#1) \end{aligned}$$

The compilation into CES-machine code for a λ-term is as follows:

Lambda Terms	Compilation
$[\![\lambda x.t]\!]_v$	$[\mathsf{Clo}([\![t]\!]_{x:v} +\!\!+ [\mathsf{Ret}])]$
$[\![M\ N]\!]_v$	$[\![N]\!]_v +\!\!+ [\![M]\!]_v +\!\!+ [\mathsf{App}]$
$[\![x]\!]_v$	$[\mathsf{Access}(n)]$ where $n = \mathsf{index}\ x\ v$
$[\![k]\!]_v$	$[\mathsf{Const}(k)]$
$[\![a + b]\!]_v$	$[\![b]\!]_v +\!\!+ [\![a]\!]_v +\!\!+ [\mathsf{Add}]$
$[\![a * b]\!]_v$	$[\![b]\!]_v +\!\!+ [\![a]\!]_v +\!\!+ [\mathsf{Mul}]$
$[\![a \leq b]\!]_v$	$[\![b]\!]_v +\!\!+ [\![a]\!]_v +\!\!+ [\mathsf{Leq}]$
$[\![\mathsf{True}]\!]_v$	$[\mathsf{True}]$
$[\![\mathsf{False}]\!]_v$	$[\mathsf{False}]$
$[\![\mathsf{If}\ t\ \{\mathsf{True} \mapsto t_0 \vert \mathsf{False} \mapsto t_1\}]\!]_v$	$[\![t]\!]_v +\!\!+ [\mathsf{If}([\![t_0]\!]_v +\!\!+ [\mathsf{Ret}], [\![t_1]\!]_v +\!\!+ [\mathsf{Ret}])]$
$[\![\mathsf{Nil}]\!]_v$	$[\mathsf{Nil}]$
$[\![\mathsf{Cons}(a, b)]\!]_v$	$[\![b]\!]_v +\!\!+ [\![a]\!]_v +\!\!+ [\mathsf{Cons}]$
$[\![\mathsf{Case}\ t\ \{\mathsf{Nil} \mapsto t_0 \vert \mathsf{Cons}\ x\ y \mapsto t_1\}]\!]_v$	$[\![t]\!]_v +\!\!+ [\mathsf{Case}([\![t_0]\!]_v +\!\!+ [\mathsf{Ret}], [\![t_1]\!]_{x:y:v} +\!\!+ [\mathsf{Ret}])]$

This translation has two slightly subtle aspects.

1. The translation is always done in a variable context, v: one starts with the empty variable context. The variable context is a stack of variable names and is used when one wants to translate a variable into its de Bruijn index: the index is just the depth of the variable in the context subscripting the translation. The de Bruijn index indicates how the value of the variable can be accessed in the environment of the CES-machine. In particular, when translating a case expression, the Cons branch requires that one adds the variables in the pattern to the context which changes how the deBruijn indexes are calculated.

2. In the translations for closures, cases, and conditionals, one must add a return instruction Ret, to the end of the code fragments within these constructs as after executing such a code fragment one must return to executing the code which followed the construct and is pushed as a continuation onto the stack.

Example 3.2 Example of compilation:

(1)

$$\begin{aligned}[\![(\lambda x.x + 1)2]\!]_{[]} &= [\mathsf{Const}(2), \mathsf{Clo}([\![(x + 1)]\!]_{[x]} +\!\!+ [\mathsf{Ret}]), \mathsf{App}] \\ &= [\mathsf{Const}(2), \mathsf{Clo}([\mathsf{Const}(1), \mathsf{Access}(1), \mathsf{Add}, \mathsf{Ret}]), \mathsf{App}]\end{aligned}$$

(2) Let us compile Ω:

$$\begin{aligned}&[\![(\lambda x.x\ x)\ (\lambda x.x\ x)]\!]_{[]} \\ &\quad= [\![(\lambda x.x\ x)]\!]_{[]} +\!\!+ [\![(\lambda x.x\ x)]\!]_{[]} +\!\!+ [\mathsf{App}] \\ &\quad= [\mathsf{Clo}([\![(x\ x)]\!]_{[x]} +\!\!+ [\mathsf{Ret}]), \mathsf{Clo}([\![(x\ x)]\!]_{[x]} +\!\!+ [\mathsf{Ret}]), \mathsf{App}] \\ &\quad= [\mathsf{Clo}([\mathsf{Access}(1), \mathsf{Access}(1), \mathsf{App}, \mathsf{Ret}]), \mathsf{Clo}([\mathsf{Access}(1), \mathsf{Access}(1), \mathsf{App}, \mathsf{Ret}]), \mathsf{App}]\end{aligned}$$

The machine Transitions in Modern SECD Machine are:

Before			After		
Code	Env	Stack	Code	Env	Stack
$\mathsf{Clo}(c') : c$	e	s	c	e	$\mathsf{Clos}(c', e) : s$
$\mathsf{App} : c$	e	$\mathsf{Clos}(c', e') : v : s$	c'	$v : e'$	$\mathsf{Clos}(c, e) : s$
$\mathsf{Access}(n); c$	e	s	c	e	$e(n) : s$
$\mathsf{Ret} : c$	e	$v : \mathsf{Clos}(c', e') : s$	c'	e'	$v : s$
$\mathsf{Const}(k) : c$	e	s	c	e	$k : s$
$\mathsf{Add} : c$	e	$n : m : s$	c	e	$(n + m) : s$
$\mathsf{Mul} : c$	e	$n : m : s$	c	e	$(n * m) : s$
$\mathsf{Leq} : c$	e	$n : m : s$	c	e	$(n \leq m) : s$
$\mathsf{True} : c$	e	s	c	e	$\mathsf{True} : s$
$\mathsf{False} : c$	e	s	c	e	$\mathsf{False} : s$
$\mathsf{If}(c_0, c_1) : c$	e	$\mathsf{True} : s$	c_0	e	$\mathsf{Clos}(c, e) : s$
$\mathsf{If}(c_0, c_1) : c$	e	$\mathsf{False} : s$	c_1	e	$\mathsf{Clos}(c, e) : s$
$\mathsf{Nil} : c$	e	s	c	e	$\mathsf{Nil} : s$
$\mathsf{Cons} : c$	e	$v_1 : v_2 : s$	c	e	$\mathsf{Cons}(v_1, v_2) : s$
$\mathsf{Case}(c_1, c_2) : c$	e	$\mathsf{Cons}(v_1, v_2) : s$	c_1	$v_1 : v_2 : e$	$\mathsf{Clo}(c, e) : s$
$\mathsf{Case}(c_1, c_2) : c$	e	$\mathsf{Nil}() : s$	c_2	e	$\mathsf{Clos}(c, e) : s$

Where $\mathsf{Clos}(c, e)$ denotes closure of code c with environment e and $e(n)$ is the n^{th}-element of the environment.

The machine is started with a code pointer and the environment and stack empty:

$$\begin{aligned} \text{Code} &= c \\ \text{Environment} &= \mathsf{Nil} \\ \text{Stack} &= \mathsf{Nil} \end{aligned}$$

The final state is reached when the code stack is empty: the answer should they be sitting on the top of the stack

$$\begin{aligned} \text{Code} &= \mathsf{Nil} \\ \text{Environment} &= \mathsf{Nil} \\ \text{Stack} &= v : \_ \qquad \text{the answer is } v! \end{aligned}$$

As an example of an evaluation in the Modern SECD machine, let us evaluate

$$\mathsf{Const}(2) : \mathsf{Clo}[\mathsf{Const}(1) : \mathsf{Access}(1) : \mathsf{Add} : \mathsf{Ret}] : \mathsf{App}$$

This, as was shown above, is the compilation of $(\lambda x.x + 1)\ 2$.

Code	Env	Stack
$\mathsf{Const}(2) : \mathsf{Clo}(c) : \mathsf{App}$	Nil	Nil
$\mathsf{Clo}(c) : \mathsf{App}$	Nil	$\mathsf{Const}(2)$
App	Nil	$\mathsf{Clos}(c, \mathsf{Nil}) : 2$
$\mathsf{Const}(1) : \mathsf{Access}(1) : \mathsf{Add} : \mathsf{Ret}$	2	$\mathsf{Clos}(\mathsf{Nil}, \mathsf{Nil})$
$\mathsf{Access}(1) : \mathsf{Add} : \mathsf{Ret}$	2	$1 : \mathsf{Clos}(\mathsf{Nil}, \mathsf{Nil})$
$\mathsf{Add} : \mathsf{Ret}$	2	$2 : 1 : \mathsf{Clos}(\mathsf{Nil}, \mathsf{Nil})$
Ret	2	$3 : \mathsf{Clos}(\mathsf{Nil}, \mathsf{Nil})$
Nil	Nil	3

where $c := \mathsf{Const}(1) : \mathsf{Access}(1) : \mathsf{Add} : \mathsf{Ret}$.

Fixed points

For a by-value machine, such as the CES-machine above, using a fixed point combinator will often cause a non-terminating behaviour. However, this can be avoided by a judicious choice of fixed point combinator. The idea is to use the fact that the machine only evaluates terms to weak head normal form, thus it is possible to avoid the undesirable behaviour of a fixed point combinator by choosing one which evaluates initially to a closure. Here is an example of one which works (see Larry Paulson's notes for example):

$$\lambda f.(\lambda a.f\ (\lambda x.a\ a\ x))\ (\lambda a.f\ (\lambda x.a\ a\ x)).$$

In fact, this is a fixed point combinator in a slightly unusual way as to show it is such we must use the η-rule:

$$\begin{array}{rl} & (\lambda f.(\lambda a.f\ (\lambda x.a\ a\ x))\ (\lambda a.f\ (\lambda x.a\ a\ x)))\ F \\ \rightarrow_\beta & (\lambda a.F\ (\lambda x.a\ a\ x))\ (\lambda a.F\ (\lambda x.a\ a\ x)) \\ \rightarrow_\beta & F\ (\lambda x.(\lambda a.F\ (\lambda x.a\ a\ x))\ (\lambda a.F\ (\lambda x.a\ a\ x))x) \\ =_\eta & F\ ((\lambda a.F\ (\lambda x.a\ a\ x))\ (\lambda a.F\ (\lambda x.a\ a\ x))) \\ \leftarrow_\beta & F\ ((\lambda f.(\lambda a.f\ (\lambda x.a\ a\ x))\ (\lambda a.f\ (\lambda x.a\ a\ x)))\ F) \end{array}$$

The extra abstraction produced by this fixed point combinator, which necessitated a use of the η-equality in the proof above, stops the CES machine from repeatedly unwrapping the fixed point leading to an an undesirable infinite behaviour.

While this works this is not very efficient! The question of how to make this more efficient has, of course, been considered as the machine has been the basis of many implementations. The technique which is often used in practical implementations is to replace a recursive definition by a closure which, when the fixed point is called, actually points back to itself: this is called "tying the knot" and gives an efficient implementation. However, it is fairly drastic as it requires a pointer modification. A less drastic approach to this is to allow the machine to call "subroutines" in which case recursion can be implemented as a direct recursive call. In effect one is "tying the knot" with such an approach but as in effect one is allowing pointers to code. To implement this one has to add to the machine a program store, Prog, and a command Call which in the machine has the transition:

Before			After		
Code	Env	Stack	Code	Env	Stack
$\mathsf{Call}(\langle f \rangle) : c$	e	s	$\mathsf{Prog}(\langle f \rangle)$	e	$\mathsf{Clos}(c, e) : s$

The called code must always end with a Ret to ensure that the code in the closure after the call is correctly initiated.

The question remains, however, whether one can implement fixed points directly in the machine. It turns out that this can also be done[2]. This does not require the use of pointers or adding a program store: one adds instead a native fixed point combinator, $\mathsf{Fix}(l)$. One then has to have a compilation to CES-machine code for this native fixed point and some additional transitions in the CES-machine. The compilation step is as follows:

$$\begin{aligned} [\![\mathsf{Fix}(\lambda f x.N)]\!]_{x'} &= \mathsf{Fix}([\![N]\!]_{x:f:x'} +\!+ [\mathsf{Ret}]) \\ [\![\mathsf{Fix}(\lambda f.N)]\!]_{x'} &= \mathsf{Fixc}([\![N]\!]_{f:x'} +\!+ [\mathsf{Ret}]) \end{aligned}$$

The two translation phrases are needed to handle the case when the function being fixed has one or more arguments and the (more unusual but possible) case when the function has no arguments. The code has to accommodate the these two cases slightly differently.

Here the translation of N assumes that the two arguments f and x are bound so the de Bruijn indexes for f and x in the code must take this into account. Notice also that the code inside the fixed point combinator must be applied to two arguments: the first is the fixed point itself and the second is the argument of the function. Here is how the machine must be modified:

Before			After		
Code	Env	Stack	Code	Env	Stack
$\mathsf{Fix}(c') : c$	e	s	c	e	$\mathsf{FixClos}(c', e) : s$
$\mathsf{App} : c$	e	$\mathsf{FixClos}(c', e') : v : s$	c'	$v : \mathsf{FixClos}(c', e') : e'$	$\mathsf{Clos}(c, e) : s$
$\mathsf{Fixc}(c') : c$	e	s	c	e	$\mathsf{FixcClos}(c', e) : s$
$\mathsf{App} : c$	e	$\mathsf{FixcClos}(c', e') : s$	c'	$\mathsf{FixcClos}(c', e') : e'$	$\mathsf{Clos}(c, e) : s$

The application for a fixed point not only applies the code but also inserts the fixed point below the argument of the application (if there is one) in the environment so that a recursive call can be serviced correctly.

Here are some questions:

1. What happens when one evaluates Ω?
2. Given a Church numeral can you translate it into an integer?
3. Can you sum the elements of a list of integers?
4. Give the λ-representation of a list of numbers can you translate it into a built-in list?
5. Can you program the factorial of a number recursively?

[2]Thanks to Nathan Harms for working this out!!!

3.2 The Krivine machine

The Krivine machine is remarkable for its simplicity. An aspect which is a bit subtle is interpreting its output: for this one must reverse compile the state of the machine into a λ-term. It performs a weak by-name reduction for pure λ-terms producing a weak head normal form. As for the CES machine it is based on a state consisting of a triple of code, environment, and stack. The environment and the stack, however, now both contain closures which makes the machine more uniform. As for the CES machine, the Krivine machine uses de Bruijn notation to facilitate compilation to code. The Krivine machine has just three instructions:

Instruction	Explanation
$\mathsf{Push}(c)$	Pushes a closure of the code c with current environment on the stack
$\mathsf{Access}(n)$	Continues with the n^{th} closure in the environment.
Grab	Moves the top value of the stack onto the environment

The compilation from de Bruijn terms is also remarkably simple – unlike for the CES machine translation no appending of code is required:

Lambda Terms	Compilation
$[\![\lambda.M]\!]$	$\mathsf{Grab} : [\![M]\!]$
$[\![M\ N]\!]$	$\mathsf{Push}([\![N]\!]) : [\![M]\!]$
$[\![\#(n)]\!]$	$[\mathsf{Access}(n)]$

Finally the machine transitions are:

Before			After		
Code	Env	Stack	Code	Env	Stack
$\mathsf{Access}(1) : c$	$\mathsf{Cls}(c', e') : e$	s	c'	e'	s
$\mathsf{Access}(n+1) : c$	$\_ : e$	s	$\mathsf{Access}(n) : c$	e	s
$\mathsf{Grab} : c$	e	$\mathsf{Cls}(c', e') : s$	c	$\mathsf{Cls}(c', e') : e$	s
$\mathsf{Push}(c') : c$	e	s	c	e	$\mathsf{Cls}(c', e) : s$

The start state for the Krivine machine has an empty environment and state with the code generated form a (closed) λ-term. The final state is reached when no more transitions can be made. The result is a (suspended) machine state which can be reverse compiled into a λ-term. There are three possible ways in which the machine can get "stuck": while doing an Access, a Grab, or one can run out of code. If one starts with a closed term, for an Access to fail a Grab must first have failed, thus, the machine must get stuck at a Grab or when the code is empty. However, it is not hard to see that the code never becomes empty. Thus, one always terminates – if indeed it does terminate – with some code applied to an environment. The environment is a list of closures – each consisting of code and environment pairs - thus, to reverse compile one recursively reverse compiles the environments into a stack of λ-terms. Then to reverse compile the top code environment pair one uses the stack of λ-terms obtained by reverse compiling the environments to substitute the access commands which are not then bound in the code reverse compiled to a λ-term The de Bruijn

index value beyond the binding depth of the term itself, indicates the depth of the term in the environment with which it should be substituted.

Here is the description of reverse compiling: one starts with the machine state $(c, e, \_)$ for which the reverse compilation is $[\![c, e]\!]^0_{\mathsf{rev}}$ as defined below.

$$\begin{array}{rcll}
[\![\mathsf{Push}(c) : c', e]\!]^i_{\mathsf{rev}} & = & [\![c', e]\!]^i_{\mathsf{rev}} \ [\![c, e]\!]^i_{\mathsf{rev}} & \\
[\![\mathsf{Grab} : c, e]\!]^i_{\mathsf{rev}} & = & \lambda.[\![c, e]\!]^{i+1}_{\mathsf{rev}} & \\
[\![\mathsf{Access}(n), e]\!]^i_{\mathsf{rev}} & = & \#(n) & n \leq i \\
[\![\mathsf{Access}(n), e]\!]^i_{\mathsf{rev}} & = & [\![e(i)]\!]^0_{\mathsf{rev}} & n > i
\end{array}$$