Abstract λ-Calculus Machines

Werner E. Kluge

Department of Computer Science
University of Kiel
D–24105 Kiel, Germany
wk@informatik.uni-kiel.de

Abstract. This paper is about fully normalizing λ-calculus machines that permit symbolic computations involving free variables. They employ full-fledged β-reductions to preserve static binding scopes when substituting and reducing under abstractions. Abstractions and variables thus become truly first class objects: both may be freely substituted for λ-bound variables and returned as abstraction values. This contrasts with implementations of conventional functional languages which realize a weakly normalizing λ-calculus that is capable of computing closed terms (or basic values) only.

The two machines described in this paper are descendants of a weakly normalizing SECD-machine that supports a nameless λ-calculus which has bound variable occurrences replaced by binding indices. Full normalization is achieved by a few more state transition rules that η-extend unapplied abstractions to full applications, inserting in ascending order binding indices for missing arguments. Updating these indices in the course of performing β-reductions is accomplished by means of a simple counting mechanism that inflicts very little overhead. Both machines realize a head-order strategy that emphasizes normalization along the leftmost spine of a λ-expression. The simpler FN_SECD-machine abides by the concept of saving (and unsaving) on a dump structure machine contexts upon each individual β-reduction. The more sophisticated FN_SE(M)CD-machine performs what are called β-reductions-in-the-large that head-normalize entire spines in the same contexts. It also employs an additional trace stack M that facilitates traversing spines in search for and contracting redices.

The paper also gives an outline of how the FN_SE(M)CD-machine can be implemented as a graph reducer.

1 Introduction

Abstract computing machines are conceptual models of program execution. They exhibit the runtime structures and the basic operating and control mechanisms that are absolutely essential to perform computations specified by particular (classes of) programming languages. They may be considered common interfaces, or

Z. Horváth et al. (Eds.): CEFP 2007, LNCS 5161, pp. 112–157, 2008.

intermediate levels of program execution, shared by a variety of real computing machines irrespective of their specific architectural features. The level of abstraction may range from direct interpretation of the constructs of a (class of) language(s) to (compiling to) abstract machine code composed of some minimal set of instructions that suffices to perform some basic operations on the runtime structures and to exercise control over their sequencing.

Our interest in abstract λ-calculus machines derives from the fact that the λ-calculus is at the core of all algorithmic programming languages, procedural or functional, as we know them today. It is a theory of computable functions that talks about elementary properties of and the application of operators to operands and, most importantly, about the role of variables in this game [Chu41, Bar84, HS86]. In its purest form it knows only three syntactical figures – variables, abstractions (of variables from expressions) and applications (of operator to operand expressions) – and a single rule for transforming λ-expressions into others. This β-reduction rule, which specifies the substitution of variables by expressions, tells us in a nutshell the whole story about computing. The runtime structures that are involved in reducing λ-expressions are shared, in one form or another, by abstract machines for all algorithmic languages, particularly in the functional domain, and so are the basic mechanisms that operate on these structures. Understanding λ-calculus machines therefore is fundamental to comprehending the why and how of organizing and performing computations by machinery.

The very first machine of this kind, which has become more or less a standard model, is the SECD-machine proposed by Landin as early as 1964 [Lan64]. It is named after the four runtime structures it employs, of which the most important ones, besides a code structure C, are an environment E and a dump D which facilitate efficient substitutions while maintaining correct binding scopes. The machine is said to be weakly normalizing, meaning that substitutions and reductions under abstractions are outlawed in order to avoid the seeming complexity of full-fledged β-reductions which would be required to resolve potential naming conflicts between free variable occurrences in arguments and variables bound by the abstractions. It is due to this restriction that the SECD-machine cannot really compute abstractions as values but must represent them as closures, i.e., as unevaluated abstractions embedded in the environments that hold instantiations of their (relatively) free variables [1].

It can justifiably be argued that this restriction, for all practical purposes, is of minor relevance if we are mainly interested in computing basic values (or ground terms) only, which is what real-life application programming overwhelmingly is all about. In fact, all implementations of functional languages are based on weakly normalizing machinery with a naive parameter passing (or substitution) mechanism, well known examples being the G-machine, the STG-machine, the Functional Abstract Machine (FAM) or the Categorial Abstract Machine (CAM) [Joh84, PeyJ92, CMQ83, CCM85/87]. Implementations of procedural languages go even one step further by demanding that functions (procedures) be

[1] We refer to a variable as being relatively free if it is free in a particular subexpression under consideration but bound higher up in a larger, surrounding expression.

legitimately applied to full sets of arguments only. Moreover, variables represent values but are not values themselves, as in the λ-calculus.

However, there are some benefits to supporting a fully normalizing λ-calulus based on a full-fledged β-reduction. Resolving naming conflicts between free and bound variable occurrences is the key to correctly performing symbolic computations as both variables and functions (abstractions) can then truly be treated as first class objects in the sense that both may be passed as function parameters and returned as function values.

This quality may be advantegeously employed, for instance, in term rewrite systems or, more specifically, in proof systems where establishing semantic equality between two terms containing free variables is an important proof tactics. Another useful application of full normalization is in the area of high-level program optimizations, e.g., by converting partial function applications into new, specialized functions with normalized bodies. Such optimizations could pay off significantly in terms of runtime performance if the specialized functions are repeatedly called in different contexts.

This paper is to show how fully normalizing abstract λ-calculus machines can be derived from standard SECD-machinery by a few minor extensions and modifications, and how these machines can be taken as blueprints for the design of equivalent graph reduction machines whose runtime efficiencies are competitive with those of its weakly normalizing counterparts.

To do so, we will proceed as follows: In the next section we will look at a very simple program to illustrate some of the shortcomings of current implementations of functional languages in order to make a case for supporting a fully normalizing λ-calculus. Section 3 introduces a normal-order SECD-machine which supports a nameless λ-calculus that has bound variables replaced by binding indices. In section 4 we will first outline the concept of head-order reductions (which is just a particular way of looking at normal-order evaluation) and then introduce in section 5 a fully normalizing FN_SECD-machine that differs from its weakly normalizing counterpart by the addition of a few more state transition rules that primarily deal with unapplied abstractions.

In section 6 we will introduce a more sophisticated FN_SE(M)CD-machine that performs what are called head-order reductions-in-the-large. It engages the dump only when entering (or returning from) the evaluation of so-called suspensions [2] and also employs an additional trace stack M for apply nodes and abstractors encountered while traversing an expression in search for β-redices. Section 7 outlines the workings of a fully normalizing graph reducer that derives from this FN_SE(M)CD-machine.

2 Some Simple Exercises in Functional Programming

To motivate what we are trying to accomplish, let's have a look at several variants of a very simple functional program, written in SCHEME [Dyb87], that exposes

[2] Loosely speaking, these are expressions embedded in their environments whose evaluation has been postponed under the normal-order strategy.

some of the problems of weak normalization. This program consists of the following two function definitions:

```
( define twice ( lambda ( f u ) ( f ( f u ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )
```

The function `twice` applies whatever is substituted for its first parameter `f` twice to whatever is substituted for its second parameter `u`, and the function `square` computes the square of a number substituted for its parameter `v`.

When applying `twice` to `square` and 2, a SCHEME interpreter returns, as one would expect,

```
( twice square 2 ) --> 16
```

i.e., the square of the square of 2. But if `twice` is just applied to either `square` or to itself, we get

```
( twice square ) --> procedure twice: expects 2 args,
                     given 1 : ( lambda(a1) ... )

( twice twice )  --> procedure twice: expects 2 args,
                     given 1 : ( lambda(a1) ... )
```

i.e., the interpreter notifies us in both cases of attempts to apply a function of two parameters to just one argument, indicating that the result is a function of one parameter that is artificially introduced as `a1`, but it cannot return a full function body in SCHEME notation.

The same happens with the application

```
( twice twice square ) --> procedure twice: expects 2 args,
                           given 1 : ( lambda(a1) ... )
```

though here `twice` is applied to two arguments, so everything should work out. However, the problem now arises in the body of `twice` where the parameter `f` is applied to just one parameter `u`, but `f` is substituted by `twice` itself, which expects two arguments. Again, the result is a function of one parameter `a1`, as one would expect, whose body cannot be made explicit.

We now slightly modify the function `twice`, turning it into curried form (i.e., into a nesting of unary functions), and see what happens then.

```
    ( define twice ( lambda ( f )
                   ( lambda ( u ) ( f ( f u ) ) ) ) )
```

When matching the curried version of `twice` by corresponding nestings of applications, as for instance in

```
    ( ( twice square ) 2 )         -->  16
```

or in

```
    ( ( ( twice twice ) square ) 2 ) -->  65536
```

we obviously get the expected results. However, when applying `twice` to two arguments, as in

```
    ( ( twice twice square ) 2 )
                    -->  procedure twice: expects 1 arg,
                         given 2 : ( lambda(a1) ... )
```

the interpreter complains about a unary function being applied to two arguments, the result of which is a function of one parameter (which is correct) whose body, again, cannot be returned in SCHEME notation.

In the following two applications, we have no mismatching arities,

```
    ( twice twice )              -->  ( lambda (a1) ... )

    ( ( twice twice ) square )   -->  ( lambda (a1) ... )
```

Here again we are only told that the result is a function of one parameter, but the function body is not disclosed.

However, what one would wish to see as output of these latter two applications, and what a fully normalizing λ-calculus would readily deliver, is something like this:

```
    ( twice twice ) --> ( lambda ( u') ( lambda ( u )
                          ( u'( u'( u'( u' u ) ) ) ) ) )

    ( ( twice twice ) square )
             --> ( lambda ( u )
                  ( * ( * ( * ( * u u ) ( * u u ) )
                  ( * ( * u u ) ( * u u ) ) ) ( .... ) ) )
```

i.e., the self-application of `twice` should return in high-level notation a function that could be called `double-twice` as it applies four times its first to the second parameter [3]. Applying this self-application to `square` should return a function of one parameter (which is expected to be substituted by a number) that is multiplied 16 times by itself. Both functions may be considered spezialized versions of the original partial applications. They may be applied in different contexts without going repeatedly through the motions of evaluating them as parts of full applications, i.e., these functions are in fact optimized.

The unfortunate state of affairs of not being able to compute functions truly as function values, let alone returning them in the above form as output, is

[3] Note that evaluating this self-application produces a naming conflict between a bound and a relatively free occurrence of the variable `u` which must be resolved by renaming either one of them as `u'`.

common to all current implementations of functional languages, e.g., HASKELL, CLEAN, ML or SCHEME [Bird98, PvE93, Ull98, Dyb87]. Little is accomplished if the programmer is just informed that the result of some computation (that generally may be rather complex) is a function, without telling what the function looks like, i.e., what exactly it computes [4]. This deficiency is a direct consequence of compiling, for reasons of runtime efficiency, programs of these languages to code of some abstract or real machine. Such code being static, it expects the right things (the objects of the computation) to be in the right places (memory locations) at the right time (or state of control). More specifically, it means that, as the above examples indicate, function (abstraction) code can execute correctly if and only if it can access at prefixed locations relative to the top of the runtime stack a full set of arguments (of the right types), i.e., an actual for each of its formal parameters. Otherwise, code execution must either be suspended until missing arguments can be picked up later on, or the user must be notified, as in the above examples, that the computation is getting stuck in a state that cannot be decompiled into a legitimate program expression.

This is to say that, in λ-calculus terminology, these languages in fact feature a weakly normalizing semantics that is more or less imposed by the constraints of compiling to static code: a function application can only be evaluated if the function's arity matches the number of arguments supplied; a partial function application may have its arguments evaluated but nothing can be done beyond that since neither substitutions under the (remaining) abstraction nor evaluation of the abstraction body are permitted.

Static code seems to leave no room for the flexibility that is required to support full normalization, in which case the code would have to deal with partial applications, i.e., with varying numbers of arguments on the stack, and with free variables (which are their own values). Also, new code would have to be generated at runtime for new functions that are being computed by application of existing ones. Though these things can be done in principle, it is generally believed that they are difficult to implement, degrading runtime performance considerably, and therefore considered a luxury that is not really needed.

However, in the following we will show that full normalization can be achieved with little effort, in terms of additional machinery, beyond what is necessary to perform weakly normalizing computations.

3 A Weakly Normalizing λ-Calculus Machine

A good starting point for the design of a fully normalizing λ-calculus machine is Landin's classical SECD-machine [Lan64]. It is an abstract applicative order evaluator that reduces λ-expressions to weak normal forms. The operating principles of this machine are based on the ideas of delayed substitutions, environments and, related to it, the notion of closures.

[4] Typed languages such as HASKELL, CLEAN or ML can at least infer the type of the resulting function which, however, is not of much help either.

The concept of delayed substitutions is to split β-reductions up into two steps that are distributed over space and time. Upon encountering β-redices, generally several in succession, the machine just collects in an environment structure the operand expressions to be substituted. All substitutions are then done in one sweep through the abstraction body by looking the operands up in the environment. Closures are special constructs that, loosely speaking, pair abstractions with the environments in which they may have to be evaluated later on.

We will first show how the SECD-machine can be modified to support normal-order evalution which guarantees termination with weak normal forms, so they exist, and then upgrade it to reduce λ-expressions to full normal forms.

3.1 A Machine-Compatible Syntax for λ-Expressions

We begin the construction of a normal-order SECD-machine with the choice of a suitable syntax for λ-expressions, taking into account that machines have a hard time dealing with variables and parentheses. We therefore use the nameless λ-calculus of deBruijn [Bru72] which replaces λ-bound variable occurrences with binding indices. We also switch to nameless abstractors Λ, replace left parentheses of applications with apply nodes @, and drop right parentheses altogether. The ensuing constructor syntax of what we in the following will refer to as the Λ-calculus thus looks like this:

$$e_\Lambda \;=_s\; \#i \mid \Lambda\, e_b \mid @\; e_f\; e_a$$

Expressions are deBruijn indices $\#i$, abstractions and normal-order applications, respectively. The apply node @ and the abstractor Λ are the constructors of this syntax.

DeBruijn indices may assume values $i \in \{\; 0, \ldots, n-1 \;\}$, where n is the number of Λ-abstractors encountered along the path from the root node of the Λ-expression down to the occurrence of the index $\#i$. The index itself measures the distance, in terms of intervening Λs, to the one that binds it (with index $\#0$ being bound to the innermost Λ).

The expressions e_f and e_a are considered operator and operand, respectively, of an application. If the operator happens to be an abstraction, then it may alternatively be referred to as the function and the operand as the argument of the application [5].

In addition to Λ-expressions, the machine also works with two syntactical constructs $[\; E\; \Lambda\, e_b \;]$ and $[\; E\; e \;]$ which respectively are called closures and suspensions. They both pair expressions with the environments in which they may have to be evaluated. The difference between the two is that closures are specifically created for abstractions that occur in operator positions of applications,

[5] It should be noted that scanning an application from left to right is equivalent to traversing in pre-order the underlying binary tree structure, i.e., the apply node at the root is inspected first, followed by operator and operand as left and right subtrees, respectively, recursively in pre-order.

whereas suspensions are created for operand expressions, including abstractions, to delay their evaluation until called for by the normal-order regime later on. Syntactically, closures are just special suspensions.

3.2 The Basics of Doing β-Reductions

A brief illustration of how β-reductions are being processed by the abstract machine we are going to design is given in fig.1. It shows how the graph representation of the nested application @ @ @ $\Lambda\Lambda\Lambda\, e_b\ e_1\ e_2\ e_3$ is step by step transformed, beginning in the upper left and following the thick arrows.

We assume that this nested application is part of a larger, surrounding expression, and that β-reductions performed in this expression have produced some

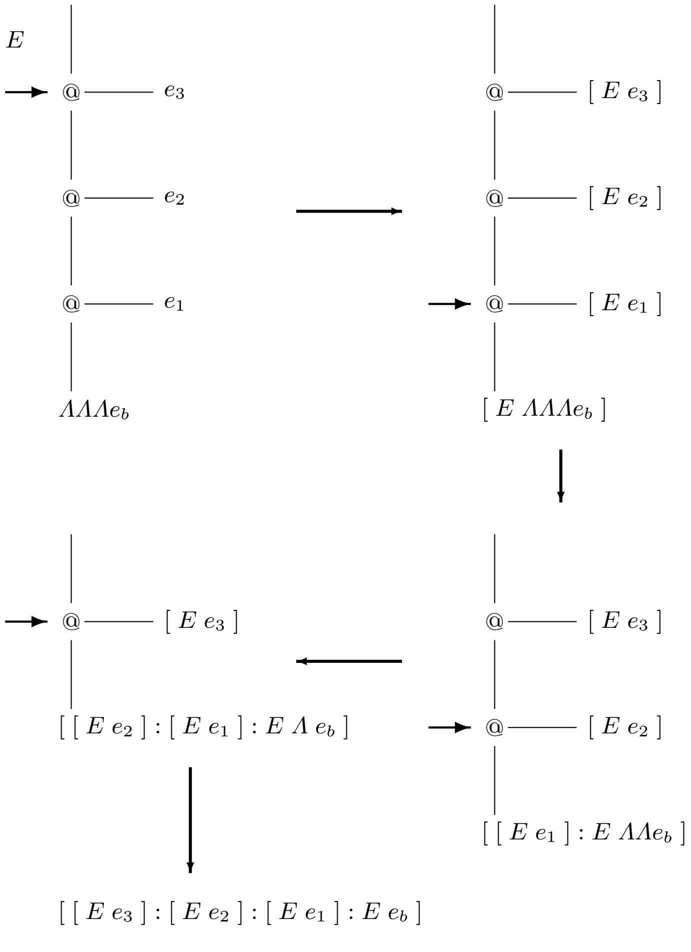

E
@ e_3
@ e_2
@ e_1
$\Lambda\Lambda\Lambda e_b$
@ $[\,E\ e_3\,]$
@ $[\,E\ e_2\,]$
@ $[\,E\ e_1\,]$
$[\,E\ \Lambda\Lambda\Lambda e_b\,]$
@ $[\,E\ e_3\,]$
@ $[\,E\ e_2\,]$
$[\,[\,E\ e_1\,] : E\ \Lambda\Lambda e_b\,]$
@ $[\,E\ e_3\,]$
$[\,[\,E\ e_2\,] : [\,E\ e_1\,] : E\ \Lambda\ e_b\,]$
$[\,[\,E\ e_3\,] : [\,E\ e_2\,] : [\,E\ e_1\,] : E\ e_b\,]$

Fig. 1. Sequence of steps that reduces a nested application @ @ @ $\Lambda\Lambda\Lambda\ e_b\ e_1\ e_2\ e_3$

environment E [6] once the focus of control has arrived at the outermost apply node under consideration, as indicated in the upper left graph by the little arrow pointing to it from the left. The entries in this environment are suspensions which may have to be substituted for deBruijn indices that occur free in the operand expressions $e_1,\ e_2,\ e_3$ and in the abstraction $\Lambda\Lambda\Lambda e_b$ in operator position.

As the focus of control moves down the spine of apply nodes, the environment E is distributed over the operand expressions $e_1,\ e_2,\ e_3$, creating suspensions in their places, and over the abstraction $\Lambda\Lambda\Lambda e_b$ in operator position, wrapping it up in a closure, as shown in the upper right graph.

It is important to note that at this point no attempts have been made to evaluate these constructs: the normal-order regime demands that, for the time being, the suspensions in operand positions be left untouched. The closure in operator position cannot be evaluated either as it would require substituting environment entries under an abstraction, which is outlawed under a weakly normalizing regime.

However, with the focus of control now pointing to the innermost apply node, we have an instance of a β-redex with an abstraction embedded in a closure in operator position and a suspension in operand position. Evaluating this application creates a new closure in its place that has one Λ removed from the abstraction and has the operand suspension prepended to the environment (denoted as $[\ E\ e_1\ ] : E$), as depicted in the graph at the lower right. Continuing in this way, the whole spine is consumed from the bottom up, resulting in a closure that has two more entries prepended to the original environment E which is now paired with an abstraction body e_b that is stripped off all Λ-abstractors (at the bottom of fig. 1). This being the case, the closure can now safely be evaluated by substituting all occurrences of deBruijn indices $\#i$ in e_b by the entries found i positions deep in the environment (counting from left to right and beginning with the index $i = 0$) as they are, i.e., without worrying about naming conflicts. We will refer to such substitutions, and in consequence also to β-reductions realized in this form, as being naive.

Note that we have chosen here the ideal case that the number of apply nodes along the spine matches the number of Λs in (or the arity of) the abstraction that is in the head of the spine, but the other cases are covered as well. If the number of apply nodes exceeds the abstraction's arity, then a shorter spine is left over with a closure as at the bottom of fig. 1 in its head. Should the arity of the abstraction exceed the number of apply nodes along the spine, i.e., we have a partial application, then we end up with a closure containing an abstraction of lesser arity that cannot be evaluated any further.

3.3 A Normal-Order SECD-Machine

The workings of an abstract machine are described by a set of machine states and a state transition function that maps (transforms) current into next states.

[6] If the application would be top level, the environment would be empty, denoted as *nil*.

A state, in turn, is described by a collection of dynamically changing data structures on which the machine operates.

The name of the SECD-machine derives from four stack-like structures that make up the machine states. These are

- a code structure C that holds Λ-expressions or fragments thereof in the order in which they need to be evaluated;
- a value stack S into which are pushed the values of expressions (or subexpressions);
- an environment structure E whose entries are suspensions that may have to be substituted for deBruijn indices that pop to the top of C;
- a dump stack D for entire machine states that are pushed and popped when entering and returning from β-reductions, respectively.

Thus, a state of the SECD-machine, to which we will also refer as a configuration, is defined by a quadruple (S, E, C, D), and the state transition function as:

$$\tau_{secd} : (\ S,\ E,\ C,\ D\ ) \ \rightarrow \ (\ S',\ E',\ C',\ D'\ ) \ \ .$$

The actual contents of the stack-like runtime structures are specified as

$$stack \ =_s \ nil \mid X \mid item : stack \ \ ,$$

where nil denotes an empty stack, X stands for one of the stack symbols $S,\ E,\ C,\ D$, and $'\ :'$ separates some specific topmost symbol or expression from the rest of the stack.

The basic operating principle of this machine is to initially set up the entire Λ-expression in the code structure C, to evaluate recursively from innermost to outermost applications popping to the top of C, and to move their values over into S, where the resulting weak normal form is recursively constructed from the bottom up.

More specifically, an application $@\ e_f\ e_a$ on top of C is rearranged in post order as $e_a : e_f : @$ to have the operand evaluated before the operator and before the entire application. Following the normal-order regime, the value of e_a must be moved into S as a suspension $[\ E\ e_a\ ]$, followed by a closure $[\ E\ e_f\ ]$ if e_f happens to be an abstraction. The applicator @ then popping to the top of C forces the evaluation of the application, consuming its components from C and S and (eventually) pushing its value into S instead.

However, evaluating β-redices takes a number of intermediate steps that involve the environment and the dump. The operand suspension is prepended to the environment carried along with the closure that contains the abstraction, and the abstraction body is in isolation set up on top of C for further evaluation in this new environment. The latter is accomplished by saving on the dump the machine state that represents the entire surrounding context of the β-redex. This context in fact constitutes the return continuation with which the

Rearranging applications on C and creating suspensions on S
(1) $(S,\ E,\ @\ e_f\ e_a : C,\ D)\ \rightarrow ([\ E\ e_a\ ] : S,\ E,\ e_f : @ : C,\ D)$

Creating closures on S for abstractions on C
(2) $(S,\ E,\ \Lambda e_b : C,\ D)\ \rightarrow ([\ E\ \Lambda e_b\ ] : S,\ E,\ C,\ D)$

Substituting deBruijn indices
(3) $(S,\ E,\ \#i : C,\ D)\ \rightarrow (\mathit{lookup}(\ \#i,\ E\ ) : S,\ E,\ C,\ D)$

Entering naive β–reductions
(4) $([\ E'\ \Lambda e_b\ ] : e_a : S,\ E,\ @ : C,\ D)\ \rightarrow (S,\ e_a : E', e_b : \mathit{nil},\ (E,\ C,\ D))$

Reducing suspensions not containing abstractions
(5) $([\ E'\ e'\ ] : S,\ E,\ C,\ D)\ |\ (\ e' \neq \Lambda e_b\ )\ \rightarrow (S,\ E',\ e' : \mathit{nil},\ (E,\ C,\ D))$

Reconstructing irreducible applications in S
(6) $(\ e_b : e_a : S,\ E,\ @ : C,\ D)\ \rightarrow (@\ e_b\ e_a : S,\ E,\ C,\ D)$

Returning from naive β-reductions
(7) $(S,\ E,\ \mathit{nil},\ (E',\ C',\ D')) \rightarrow (S,\ E',\ C',\ D')$

Fig. 2. The complete set of state transition rules for the normal-order SECD machine

computation must continue once evaluation of the β-redex is completed, whereupon its value ends up on S and the code structure C becomes empty.

The details of how this machine works are specified by the set of state transition rules given in fig. 2, which realizes the state transition function τ_{secd}. They are listed in the order in which they must be matched against actual machine states.

Rules (1) to (3) identify the machine configurations that have the three syntactical figures of legitimate Λ-expressions appear on top of the code structure C. Rule (1) splits an application up into its three components which are rearranged so that the apply node is squeezed underneath the operator, whereas the operand is embedded in a suspension that is pushed into S. Rule (2) wraps an abstraction up in a closure that is pushed into S. A deBruijn index on top of C accesses, by application of rule (3), the i-th entry relative to the top of the environment E, using a function *lookup*, and pushes it into S, which realizes the substitution that completes a naive β-reduction.

Rule (4) enters a (naive) β-reduction: an applicator @ on top of C in conjunction with a closure on S has the body of the abstraction isolated for evaluation in C together with its environment on E, while the current environment and the current code structure, i.e., the calling context, are saved as return continuation on the dump. The operand retrieved from underneath the closure in S, which is bound to be a suspension, is prepended as a new entry to what has now become the active environment. Evaluating an abstraction body involves traversing it step by step from C to S, thereby substituting deBruijn indices by environment entries or calling for other (naive) β-reductions. Completing this traversal is signified by an empty code structure, at which point rule (7) is called to return to the context saved on the dump.

Isolating in C the body of an abstraction to be evaluated and in E the environment in which evaluation must take place while saving the surrounding context on the dump is a measure that ensures substitution of deBruijn indices in exactly the intended binding scope – the abstraction body – by suspensions that belong to just the relevant environment.

Rule (5) takes care of suspensions that contain expressions other than abstractions, i.e., primarily applications but also deBruijn indices. They are set up for evaluation in basically the same way as by rule (4): the expressions are isolated in C together with the corresponding environments in E, and the surrounding contexts are saved on the dump.

And finally, rule (6) reconstructs from the components spread out over C and S irreducible applications in S.

An initial machine state has the entire expression to be reduced set up in the code structure C, with all other structures empty, and the terminal state, so it exists, has its weak normal form set up in the value stack S, while all other structures are empty.

The machine stops in such a state since none of the rules of fig. 2 matches.

It has to be well understood that this machine can reduce only closed λ-expressions. This is due to the fact that deBruijn indices, by definition, cannot occur free anywhere in the expression and that therefore legitimate reducible expressions can only be top-level applications of closed abstractions to closed abstractions, as a consequence of which the resulting weak normal forms can only be abstractions embedded in closures (which syntactically are indistinguishable from suspensions).

3.4 Reducing Step by Step a Simple *Λ*-Expression

As an illustration of how this normal-order SECD-machine works, lets have a look at the sequence of machine states in fig. 3 that it brings about when reducing the Λ-expression

$$@\ @\ \Lambda\,\#0\ \Lambda\,\#0\ @\ \Lambda\,\#0\ \Lambda\,\#0$$

to its weak normal form $\Lambda\,\#0$ (which is also its full normal form).

The initial stack configuration at the top of fig. 3 has the entire expression set up in the code structure C while all other structures are empty. This expression being an application, rule (1) takes over to enclose the operand expression in a suspensions that is pushed into S, and the apply node is squeezed underneath the operator in C. The operator thus exposed as the next expression that must be taken care of again is an application which calls once more for rule (1), yielding the third stack configuration from the top. The abstraction now on top of C is by rule (2) wrapped up in a closure and pushed as value into S, thus bringing the inner apply node to the top of C, its operand being underneath the operator in S (fourth configuration).

$$\begin{array}{r@{\;|\;}l}
nil & S \\
nil & E \\
@\,@\,\Lambda\,\#0\,\Lambda\,\#0\ @\,\Lambda\,\#0\,\Lambda\,\#0 : nil & C \\
nil & D
\end{array}$$

$$\text{Rule 1} \quad \Downarrow$$

$$\begin{array}{r@{\;|\;}l}
[\,nil\ @\,\Lambda\,\#0\,\Lambda\,\#0\,] : nil & S \\
nil & E \\
@\,\Lambda\,\#0\,\Lambda\,\#0 : @ : nil & C \\
nil & D
\end{array}$$

$$\text{Rule 1} \quad \Downarrow$$

$$\begin{array}{r@{\;|\;}l}
[\,nil\ \Lambda\,\#0\,] : [\,nil\ @\,\Lambda\,\#0\,\Lambda\,\#0\,] : nil & S \\
nil & E \\
\Lambda\,\#0 : @ : @ : nil & C \\
nil & D
\end{array}$$

$$\text{Rule 2} \quad \Downarrow$$

$$\begin{array}{r@{\;|\;}l}
[\,nil\ \Lambda\,\#0\,] : [\,nil\ \Lambda\,\#0\,] : [\,nil\ @\,\Lambda\,\#0\,\Lambda\,\#0\,] : nil & S \\
nil & E \\
@ : @ : nil & C \\
nil & D
\end{array}$$

$$\text{Rule 4} \quad \Downarrow$$

$$\begin{array}{r@{\;|\;}l}
[\,nil\ @\,\Lambda\,\#0\,\Lambda\,\#0\,] : nil & S \\
[\,nil\ \Lambda\,\#0\,] : nil & E \\
\#0 : nil & C \\
(nil,\ @ : nil,\ nil) & D
\end{array}$$

$$\text{Rule 3} \quad \Downarrow$$

$$\begin{array}{r@{\;|\;}l}
[\,nil\ \Lambda\,\#0\,] : [\,nil\ @\,\Lambda\,\#0\,\Lambda\,\#0\,] : nil & S \\
[\,nil\ \Lambda\,\#0\,] : nil & E \\
nil & C \\
(nil,\ @ : nil,\ nil) & D
\end{array}$$

$$\text{Rule 7} \quad \Downarrow$$

Fig. 3. Reducing step by step the expression $@\,@\,\Lambda\,\#0\,\Lambda\,\#0@\,\Lambda\,\#0\,\Lambda\,\#0$ on the SECD-machine

$$\begin{array}{r}
[\,nil\ \varLambda\,\#0\,] : [\,nil\ @\,\varLambda\,\#0\,\varLambda\,\#0\,] : nil \mid S\\
nil \mid E\\
@ : nil \mid C\\
nil \mid D
\end{array}$$

Rule 4 $\Downarrow$

$$\begin{array}{r}
nil \mid S\\
[\,nil\ @\,\varLambda\,\#0\,\varLambda\,\#0\,] : nil \mid E\\
\#0 : nil \mid C\\
(nil,\ nil,\ nil) \mid D
\end{array}$$

Rule 3 $\Downarrow$

$$\begin{array}{r}
[\,nil\ @\,\varLambda\,\#0\,\varLambda\,\#0\,] : nil \mid S\\
[\,nil\ @\,\varLambda\,\#0\,\varLambda\,\#0\,] : nil \mid E\\
nil \mid C\\
(nil,\ nil,\ nil) \mid D
\end{array}$$

Rule 7 $\Downarrow$

$$\begin{array}{r}
[\,nil\ @\,\varLambda\,\#0\,\varLambda\,\#0\,] : nil \mid S\\
nil \mid E\\
nil \mid C\\
nil \mid D
\end{array}$$

Rule 5 $\Downarrow$

$$\begin{array}{r}
nil \mid S\\
nil \mid E\\
@\,\varLambda\,\#0\,\varLambda\,\#0 : nil \mid C\\
(\ nil,\ nil,\ nil\ ) \mid D
\end{array}$$

Rule 3 $\Downarrow$

... and so on ...

Fig. 3. (*continued*)

At this point rule (4) detects a β-redex. It removes both the operator closure and the operand suspension from S, isolates the body #0 of the abstraction in C, and also prepends the operand suspension to the empty environment *nil* carried along with the closure, which now becomes active. The old environment and the remaining code structure C, i.e., the outermost apply node, are saved on the dump (fifth configuration from the top). Evaluating the deBruijn index #0 in C calls for rule (3), which copies the environment entry at position 0 relative to its top, which is the suspension [*nil* $\Lambda\,\#0$], on top of S, leaving the control structure C empty (last configuration of fig. 3).

Having thus completed the evaluation of the operator expression of the outermost application, the machine returns, by rule (7), to the surrounding context to continue with the evaluation of the outermost application (top configuration of fig. 3). Going basically through the same motions, it arrives, after three more steps, at a configuration that has the operand of the outermost application enclosed in a suspension set up in S, with all other structures empty. As this suspension contains an application, it is intercepted by rule (5) to enforce its evaluation as well. In doing so, the machine creates a new context which has the application @ $\Lambda\,\#0$ $\Lambda\,\#0$ set up in C and the associated empty environment in E, just as before starting the evaluation of the entire expression. After performing the same four steps that reduced the identical operator expression, the machine terminates with the closure [*nil* $\Lambda\,\#0$] in S and all other structures empty.

4 Toward Fully Normalizing λ-Calculus Machines

Upgrading a weakly to a fully normalizing λ-calculus machine requires (the equivalent of) full-fledged β-reductions to preserve the functional property of the λ-calculus when substituting and reducing under abstractions. A clever implementation that can be mechanically executed almost as efficiently as naive substitutions may be obtained by taking advantage of a few more properties of the λ-calculus beyond the β-reduction rule itself that are well covered in standard textbooks [Bar84, HS86]. They are briefly reviewed in the following subsection.

4.1 β-Reduction, η-Extension, β-Distribution and Head (Normal) Forms

In the nameless Λ-calculus that is of interest here, deBruijn indices measure distances, in terms of numbers of intervening Λs, between the syntactical positions of their occurrences and the Λ-abstractors that bind them. Full-fledged β-reduction requires updating them whenever the number of Λs in between changes. More specifically, when removing intervening Λs, the indices must be decremented, and when squeezing additional Λs in between, the indices must be incremented accordingly.

Consider as a small example that may help to illustrate how this works the expression [7]

$$\Lambda_2\,@\;\Lambda_1\Lambda_0\,@\;\#1\;\#2\;\Lambda_4\,\#1\quad .$$

In the body of the abstraction $\Lambda_1\Lambda_0\,@\;\#1\;\#2$ the indices #1 and #2 are bound to Λ_1 and Λ_2, respectively, the index #1 occurs free in the abstraction $\Lambda_4\,\#1$ but is also bound to Λ_2; there are no indices that are bound to Λ_0 and Λ_4.

[7] The subscripts attached to the Λs merely serve to facilitate explaining which deBruijn index is bound to which abstractor.

This expression evaluates in two β-reduction steps as follows:

$$\Lambda_2 \,@\, \Lambda_1 \Lambda_0 \,@\, \#1\ \#2\ \Lambda_4\,\#1 \ \ \rightarrow_\beta\ \ \Lambda_2 \Lambda_0 \,@\, \Lambda_4\,\#2\ \#1 \ \ \rightarrow_\beta\ \ \Lambda_2\Lambda_0\,\#1$$

Reducing, in the first step, the outer application substitutes the abstraction $\Lambda_4\,\#1$ for the index #1 in the body of the abstraction $\Lambda_1\Lambda_0 \,@\, \#1\ \#2$, thereby removing the abstractor Λ_1 and decrementing the original index #2 as the distance to the binding Λ_2 is now one less. However, the index in the body of $\Lambda_4\,\#1$ must be incremented since crossing the abstractor Λ_0 increases the distance to the binding Λ_2 by one.

The second step β-reduces the remaining application. As the abstractor Λ_4 does not bind anything, the operand #1 is simply consumed, but the index #2 in the abstraction body is decremented to #1 since the disappearance of the abstractor Λ_4 has shortened by one the distance to the binding Λ_2.

The troublesome part about performing β-reductions in this way is that deBruijn indices may have to be counted up and down several times, as may be illustrated by the following example:

$$@\,@\,@\,\Lambda\Lambda\Lambda\,@\,\#2\,@\,\#1\,\#0\ \ \#3\ \ \#2\ \ \#1$$

(here it is assumed that the indices #3, #2, #1 in operand positions of the three nested outer applications are bound by Λ-abstractors somewhere in a surrounding expression). Reducing these applications step by step from innermost to outermost yields:

$$@\,@\,@\,\Lambda\Lambda\Lambda\,@\,\#2\,@\,\#1\,\#0\ \ \#3\ \#2\ \#1\ \ \rightarrow_\beta$$
$$@\,@\,\Lambda\Lambda @\,\#5\,@\,\#1\,\#0\ \ \#2\ \#1\ \ \rightarrow_\beta\ \ @\,\Lambda @\,\#4\,@\,\#3\,\#0\ \ \#1\ \ \rightarrow_\beta\ \ @\,\#3\,@\,\#2\,\#1$$

It is interesting to note that the operand indices are in their places of substitution in the abstraction body first stepped up by the number of Λs whose scopes are being penetrated, but that these indices are decremented again as the Λs are being consumed by subsequent β-reductions, with the net effect that they have not changed at all after all β-reductions are done. Needless to say that this is a special property of full applications which has in fact already been exploited in the weakly normalizing machine of the preceding section.

However, this example also tells us that when β-reducing step by step a partial application, free occurrences of deBruijn indices in operand expressions are, after all redices are done, in their places of substitution effectively stepped up by the number of Λs remaining, i.e., by the arity of the resulting abstraction.

More specifically, a partial application of the general form

$$\underbrace{@\ldots @}_{k}\ \underbrace{\Lambda\ldots\Lambda}_{n} e_b\ \ e_1\ \ldots\ e_k \mid k < n$$

β-reduces to an $(n-k)$-ary abstraction that has all occurrences of the deBruijn indices $\#(n-1)\ldots\#(n-k)$ in e_b substituted by the operands $e_1\ldots e_k$, respectively, in which all occurrences of (relatively) free deBruijn indices are incremented by $(n-k)$. In the special case that $k = n$, i.e., we have a full application as above, the original indices remain unchanged. Of course, all free occurrences of deBruijn indices in the original n-ary abstraction must be decremented by k.

This leads us to conclude that if we can find a way of doing these k β-reductions in one conceptual step, a lot of superfluous index updates could be spared.

As a first step toward this end, we make use of η-extensions as an elegant way of minimizing the number of updates on deBruijn indices when reducing partial applications. η-extension derives from the semantic equivalence

$$@\, e_0\, e_1 \;=\; @\, \Lambda @\, e_0^{(+1)}\, \#0 \;\; e_1 \quad ,$$

where the superscript on $e_0^{(+1)}$ denotes the addition of 1 to all free occurrences of deBruijn indices in e_0, since an additional abstractor Λ has been squeezed between them and the binding Λs that may be found in a larger, surrounding expression. This equivalence also implies that

$$e_0 \;=\; \Lambda\, @\, e_0^{(+1)}\, \#0 \quad .$$

More generally, when η-extending an abstraction k-fold, we get

$$e \;=\; \underbrace{\Lambda \ldots \Lambda}_{k} \underbrace{@ \ldots @}_{k}\; e^{(+k)}\; \#(k-1) \ldots \#0 \quad .$$

This semantic equivalence may be readily employed to turn partial into full applications that can be reduced by a weakly normalizing machine. All that needs to be done is to extend a partial application by as many applications to deBruijn indices in ascending order as there are missing operands, and to put in front of this extended application the same number of Λ-abstractors:

$$\underbrace{@ \ldots @}_{k}\; \underbrace{\Lambda \ldots \Lambda}_{n} e_b\; e_{k-1}\; \ldots e_0 \;=$$
$$\underbrace{\Lambda \ldots \Lambda}_{n-k} \underbrace{@ \ldots @}_{n-k} < \underbrace{@ \ldots @}_{k}\; \underbrace{\Lambda \ldots \Lambda}_{n} e_b\; e_{k-1} \ldots e_0 >^{+(n-k)}\;\; \#(n{-}k{-}1) \ldots \#0 \quad .$$

(the construct $< \cdots >^{+(n-k)}$ denotes incrementation by $(n-k)$ of all free occurrences of deBruijn indices in the expressions within the brackets.)

The weakly normalizing SECD-machine augmented by an appropriate mechanism for such η-extension-in-the-large can thus be made to reduce, under an $(n{-}k)$-ary abstraction, a body composed of the application of an n-ary abstraction to n operand expressions of which the outermost $(n-k)$ are deBruijn indices from the interval $\#0 \ldots \#(n-k-1)$. It creates an environment for the evaluation of the abstraction body e_b which substitutes the indices $\#(n-1) \ldots \#(n-k)$ by the expressions $e_{(k-1)} \ldots e_0$ (with updated indices) and the indices $\#(n-k-1) \ldots \#0$ by themselves.

As a second step, we will make use of the fact that β-redices can be distributed over the components of an abstraction body that is itself an application. For the simple case of distributing just one β-redex we have

$$@\, \Lambda\, @\, e_a\, e_b\;\; e_1 \;=\; @\, @\, \Lambda\, e_a\, e_1\; @\, \Lambda\, e_b\, e_1 \quad .$$

This may be generalized for n nested redices as

$$\underbrace{@\ldots@}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n} @\, e_a\, e_b\, e_1 \ldots e_n \ = \\ @\ \underbrace{@\ldots@}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n} e_a\ e_1 \ldots e_n\ \underbrace{@\ldots@}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n} e_b\ e_1 \ldots e_n \ \ ,$$

which we may call a β-distribution-in-the-large. By pushing β-redices in this way recursively in front of the subexpressions of an abstraction body, β-reductions may be delayed until and performed only when and where they are actually needed.

As a third step, we combine both η-extensions-in-the-large and β-distributions-in-the-large with a suitable reduction strategy. It may be derived from looking at the syntax of Λ-expressions from a particular perspective that emphasizes what are called head forms:

$$h \mid t \ =_s\ \#i \mid \underbrace{\Lambda\ldots\Lambda}_{n}\underbrace{@\ldots@}_{r}\ h\, t_1 \ldots t_r$$

A head form generally is a (nested) application of a single head expression h to some $r \geq 0$ tail expressions $t_1 \ldots t_r$ which is preceded by some $n \geq 0$ abstractors. Heads and tails are recursively constructed in the same way, i.e., they all have head forms as well. Trivial head expressions are deBruijn indices $\#i$. If the outermost head expression h is a deBruijn index, then we have a head-normal form.

Occurrences of deBruijn indices in Λ-expressions must always be smaller than the total number of Λs preceding them, i.e., there is no notion of such indices being free in the entire head form. However, we may consider indices as being free if they are bound to the outermost leading sequence of Λs because then they may be passed around and updated by β-reductions but they never get substituted by anything else.

Following a normal-order regime, the reduction strategy that lends itself directly to head forms is called head-order reduction as it emphasizes reductions in the head: It first reduces the head expression to head-normal form and then recursively all remaining tails to head normal forms as well, thus eventually arriving at a full normal form of the entire expression, provided the whole process terminates after finitely many β-reductions. The significance of this heads-first strategy derives from the fact that an expression cannot have a full normal form without having a head normal form, which should therefore be determined before evaluating the tails.

4.2 Head-Order Reduction

In this subsection we are going to illustrate, by means of the graphical representation of a typical head form as in fig. 4, how head-order reductions can be organized, closely following an earlier proposal by Berkling [Ber86] [8].

[8] The contents of this subsection are in large parts adopted from the author's monograph on Abstract Computing Machines[Kge05].

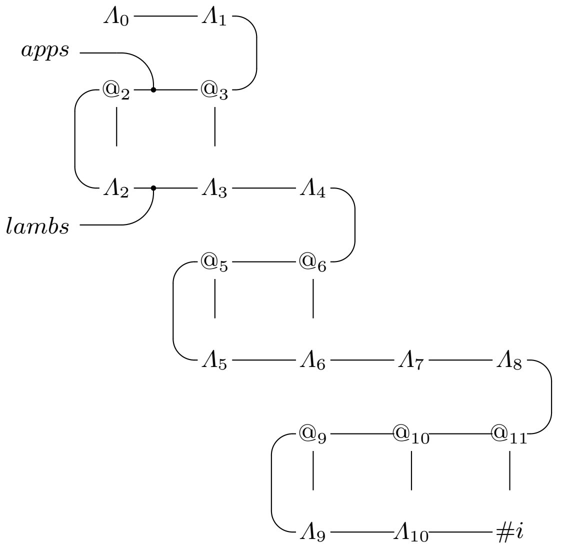


Fig. 4. A typical head form of a λ-expression

Head and tail expressions obviously are the operators and operands (depicted by the downward pointing thin lines), respectively, of applications. On the path from the root node of this graph down to the head index $\#i$ we find alternately only sequences of Λ-abstractors and sequences of applicators @, to which we will refer as *lambs* and *apps* sequences, respectively, and to the entire path as the (leftmost) spine of the head form. All tails along this spine have recursively head forms, or are spines, of their own.[9]

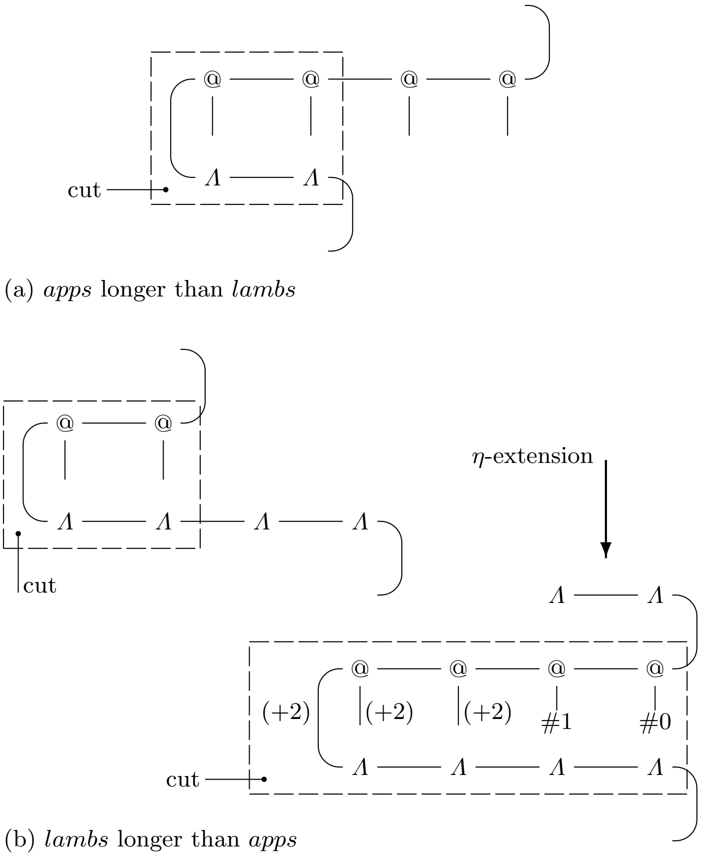
A section of the spine headed by a *lambs* sequence of length n is in fact a curried n-ary abstraction whose body stretches over the entire remaining spine, i.e., the spine of fig. 4 includes four abstractions nested inside each other.

Normal order reduction as effected by the applicator @ demands that β-redices be reduced systematically from top to bottom along such spines until no more β-redices are left, i.e., the spine features a sequence of leading Λs followed by a sequence of applicators (which may be empty) followed by a head index bound by one of the leading Λs, in which case we have arrived at a head-normal form.

Looking at the meander-like structure of the spine in fig. 4, β-redices can be easily identified in the left-hand corners that connect *apps* and *lambs* sequences and thus pair innermost apply nodes with outermost abstractors. However, rather than actually performing these β-reductions step by step from left to right, the idea of head-order reduction is to take largest possible chunks of β-redices, which we'll call cuts, out of such corners and to distribute them over the head and tail expressions of the *apps* sequence that follows next along the spine, using β-distributions-in-the-large as outlined in the preceding subsection.

[9] Λ-nodes and apply nodes are in this graph enumerated so that one can follow up more easily on what is ending up where when reducing this spine.

Just what these cuts are depends on the relative lengths of the *apps* and *lambs* sequences involved, as depicted in fig. 5 below.



(a) *apps* longer than *lambs*

(b) *lambs* longer than *apps*

Fig. 5. Taking cuts off left-hand corners

The upper part (a) shows the easier case with an *apps* sequence that has at least the same length as the *lambs* sequence. Here we have a full application as the cut matches each abstractor with an apply node.

The lower part (b) shows a corner in which the *lambs* sequence is longer than the *apps* sequence, i.e., we have a partial application that β-reduces to a new abstraction of lesser arity, which would be 2 in the particular case. This can be accomplished by means of an η-extension-in-the-large, as also introduced in the preceding subsection, that transforms the entire *apps* – *lambs* corner into a full application. The added apply nodes have the deBruijn indices #0 and #1 in their tails, and all free occurrences of deBruijn indices in the head and the tails of the original *apps* sequence are stepped up by 2, as annotated at the respective edges, to account for the two Λ-nodes introduced by the η-extension.

Inspecting the spine of fig. 4, we note that this head form includes three *apps* − *lambs* corners, of which the upper two are partial applications that must be η-extended before β-distributing them over the branches of the spine. Fig. 6 below illustrates how this is done.

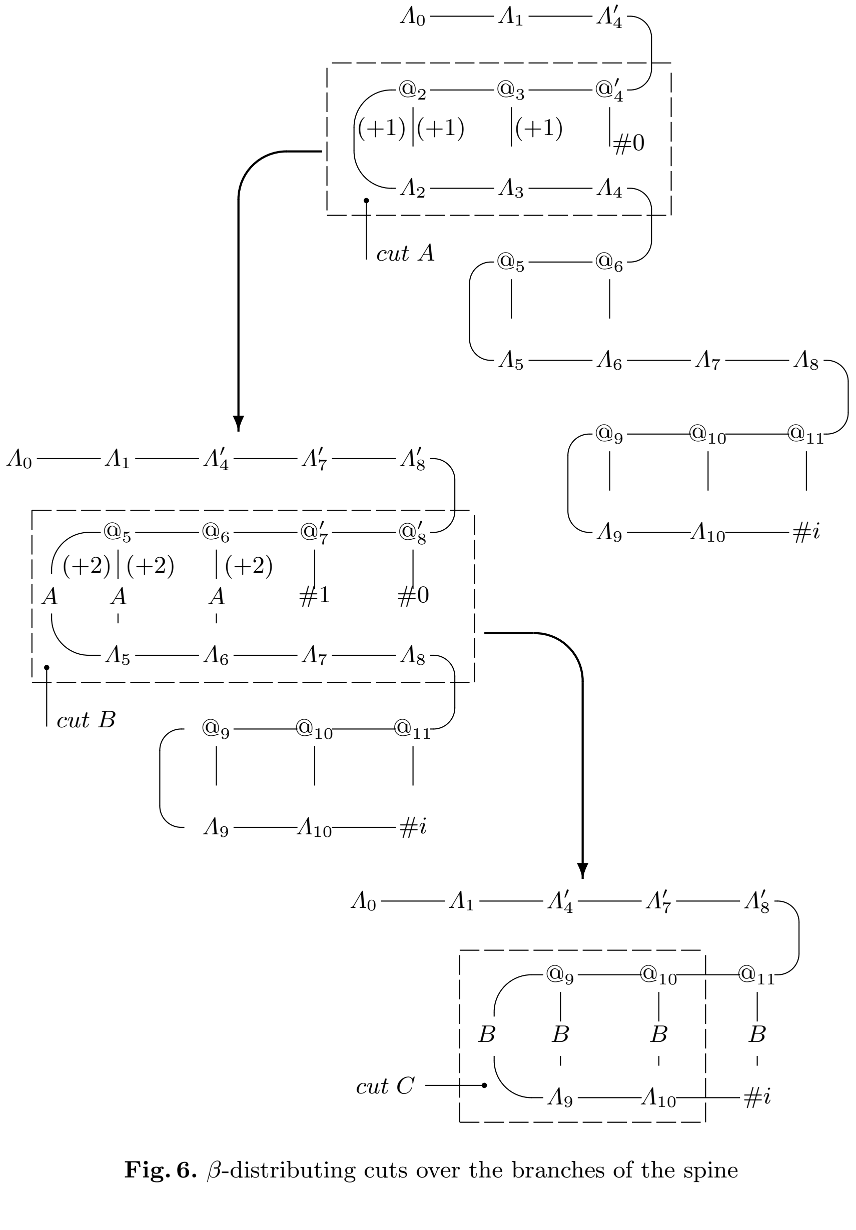


Fig. 6. β-distributing cuts over the branches of the spine

Proceeding from top to bottom along the spine, the first corner that is being encountered must be η-extended by one $\Lambda \,|\, @$ pair to obtain a cut A that represents a full application (the graph in the upper part of the figure). Distributing this cut over the next corner of the spine squeezes it in front of the tails and the head of its *apps*-sequence. This corner must be η-extended by two $\Lambda \,|\, @$ pairs to form another cut B for a full application (the graph in the middle of the figure), which in turn is β-distributed over the head and the tails of the remaining corner of the spine (the graph at the bottom). This corner constitutes a full application as it is, forming a cut C. This cut is trivially distributed just in front of the head index $\#i$, i.e., it remains in place.

If in cut C we now expand the copy of cut B that makes up its left-hand corner and, likewise, in cut B expand the copy of cut A on the left, we obtain the spine shown in fig. 7 below.

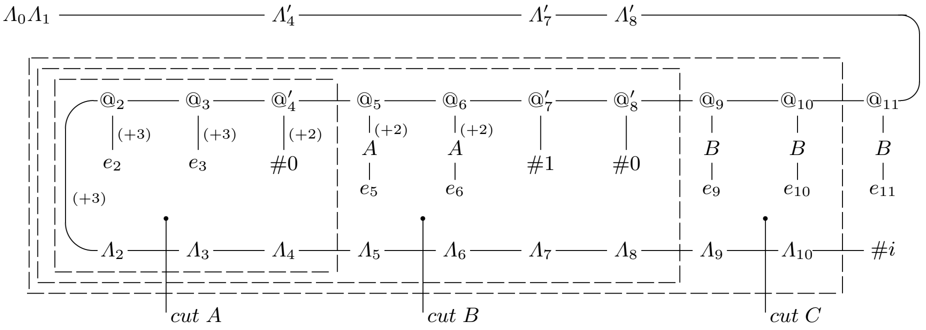


Fig. 7. The spine emerging from the one of fig. 4 after having completed all η-extensions and β-distributions-in-the-large

This spine features a leading *lambs*-sequence to which have been lifted the Λ-abstractors that have been introduced by η-extensions. It is followed by a single left-hand corner that connects an *apps* sequence of length 10 with a *lambs* sequence of length 9, i.e., we have in fact unfolded, by means of repeated η-extensions and β-distributions (...-in-the-large) what was the original cut C to nine β-redices. This new cut C includes cut B which, in turn, includes cut A [10].

Having thus straightened the original spine, we can finally contract, in one conceptual step to which we may refer as β-reduction-in-the-large, all β-redices of the original spine that have now accumulated in a single cut C, thereby completely consuming it. The resulting reductum depends on the deBruijn index $\#i$ in the head of the spine, which happens to be the entire body of the abstraction formed by the *lambs*-sequence preceding it.

If this index is smaller than 9, it is bound to a Λ within the preceding *lambs* sequence, which means that the abstraction is in fact a selector function that picks from the *apps* sequence the tail of the apply node that in the graph is

[10] Note that the apply and Λ nodes that have been introduced by η-extensions are annotated as primed and receive the same indices as the corresponding Λs in the original *lambs* sequences.

opposite to the Λ to which the index is bound. For instance, an index $i = 1$ that is bound to Λ_9 selects the tail of $@_9$, or an index $i = 4$ is bound to Λ_6 and thus returns the tail of $@_6$.

These tails are substituted in the head position of the spine that is left over after the cut C has disappeared, which is just the leading *lambs* sequence $\Lambda_0 \Lambda_1 \Lambda'_4 \Lambda'_7 \Lambda'_8$ followed by the apply node $@_{11}$ whose tail remains intact.

This process of η-extensions, β-distributions and β-reductions (-in-the-large) repeats itself in the head thus expanded until the head position is occupied by an index bound to one of the Λs of the leading *lambs*-sequence, i.e., the spine has become head-normalized.

This is the case if in the original spine of fig. 4 the index was bound either to one of the leading Λs, say $i = 10$, or to one of the unapplied Λs that gave rise to η-extensions, say $i = 6$.

In the former case, the head index is bound to Λ_0 and must remain so after cut C in fig. 7 has been completely β-reduced, i.e., the resulting index should be $i = 4$. We can easily convince ourselves that this is indeed so: there are nine intervening Λs that do disappear due to these β-reductions, decrementing the head index to $i = 1$, but three Λs have been squeezed in between due to η-extensions, resulting in the index $i = 4$.

In the latter case, the original index $i = 6$ is bound to Λ_4, which selects the index $i = 2$ (i.e., $i = 0$ incremented by 2) as the tail of $@'_4$, which in turn is bound to Λ'_4 in what has become the expanded leading *lambs* sequence.

The cuts that build up along the spine in fact define an environment, just as we know it from the SECD-machine, in which the head expression is to be evaluated. This environment just keeps expanding as long as there are *apps–lambs* corners left to be distributed down the spine. With one large *apps–lambs* corner remaining that has accumulated, in nested form, all the others that were preceding it, we have a single contiguous environment. Depending on its value, the head index defines either a single access into this environment to retrieve a tail expression that must be substituted in the head, generally leading to more β-reductions along the spine, or it is bound by one of the Λs of the resulting leading *lambs* sequence, in which case we are done with the head, having arrived at a head-normal form, and may turn to the tails, if there are any left, and recursively reduce them in head-order as well.

The tails of head normal forms are generally unevaluated expressions preceded by cuts, or by their environments, that are equivalent to the suspensions as we know them from the SECD-machine.

5 The FN_SECD-Machine

The runtime structures and the basic mechanisms of the weakly normalizing SECD-machine, not very surprisingly, can be employed in a fully normalizing machine as well. We definitely need a code structure C, an environment that holds suspensions $[\ E\ e\ ]$, some stack S that temporarily holds intermediate values, basically again suspensions but also deBruijn indices that are bound by leading

Λs. Stack S also serves as the destination of full normal forms. Beyond that, it is expedient to include a dump as well that keeps track of nested β-distributions and η-extensions, accommodating the respective return continuations.

The machine must also include an efficient η-extension mechanism that does the equivalent of generating as arguments for unapplied Λs deBruijn indices and of updating those introduced by earlier η-extensions, as outlined in subsection 4.2.

5.1 The Unapplied Lambdas Count

The basic idea of how η-extensions and the ensuing updates on deBruijn indices can be done almost effortlessly may be inferred from a close look at the spine of fig. 7.

We note that after the first η-extension that leads to cut A the deBruijn index in the tail of the apply node $@'_4$ receives the value #0. When doing the second η-extension that brings about cut B, the tails of the new apply nodes $@'_7$ and $@'_8$ receive the indices #1 and #0, respectively, and the index in the tail of $@'_4$ is stepped up by 2, which equals the number of Λs that have been squeezed in between.

Rather than updating in this way earlier deBruijn indices whenever another η-extension must be done along the spine, the very same index values may be obtained by the following method that is decidedly simpler to implement and more efficient to execute [Trou93]:

- The number of unapplied Λs introduced by η-extensions while proceeding from top to bottom along the original spine is kept track of in a count variable ULC (which stands for Unapplied Lambdas Count), beginning with the value 0 (though any other non-negative integer value could be chosen as well);
- The tails of the apply nodes introduced by η-extensions are filled with ULC values rather than deBruijn indices in monotonically ascending order;
- When needed, the correct deBruijn indices may be obtained by subtracting from the current value of the ULC counter the ULC values actually found in the η-extended tails (which may be the same or lower).

The interesting properties about this method are that the ULCs put into the η-extended tails are invariant against further η-extensions down the spine, that these values can be generated by a simple counting mechanism, and that correct index values can be calculated by a single integer subtraction, thus minimizing the effort of manipulating them.

However, in order to treat all deBruijn indices, including those that are bound by what originally were the leading Λs of the spine, in a uniform way, these unapplied abstractors must be η-extended as well.

These extensions add another (innermost) cut L to the spine of fig. 7, yielding the spine depicted in fig. 8. It has the tails of cut L filled with the ULC values 1 and 2, followed by the value 3 in the η-extended cut A and by the values 4 and

5 in the η-extended cut B. The ULC values after completion of the cuts L, A, B and C are also shown at the bottom.

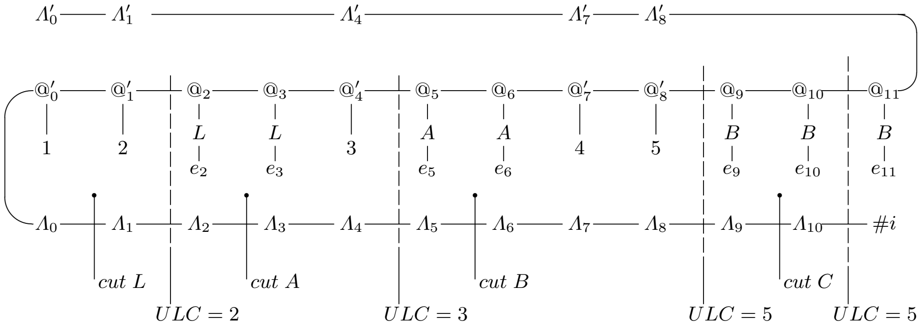


Fig. 8. The spine of fig. 7, η-extended by another cut L for the leading Λs, and showing ULC values replacing all η-extended deBruijn indices

To exemplify calculation from ULCs of correct deBruijn indices, consider environment accesses with the head indices #3, #6 and #10, all of which are bound by unapplied Λs. Index #3 picks the tail of the η-extended apply node $@'_7$, i.e., the ULC value 4. Correcting it with the ULC value 5 reached after having flattened the entire spine yields the deBruijn index #1; likewise the head index #6 selects the value 3 from the tail of $@'_4$ and, after subtracting it from the ULC-value 5, returns the deBruijn index #2. In both cases we obtain exactly the same deBruijn indices as would be selected from the spine of fig. 7. And finally, index #10 which was bound to Λ_0 in the original spine selects 1 from the tail of $@'_0$. Upon subtracting it from the ULC value 5 we get the deBruijn index #4 which remains bound by Λ'_0 in the emerging leading *lambs* sequence $\Lambda'_0\Lambda'_1\Lambda'_4\Lambda'_7\Lambda'_8$ [11].

5.2 The State Transition Rules

The state description of the FN_SECD-machine differs from that of the ordinary SECD-machine only in the addition of the unapplied lambdas count ULC as a plain variable u, i.e., the state transition rules specify mappings of the form:

$$\tau_{fn\_secd} : (S,\ E,\ C,\ D,\ u)\ \rightarrow\ (S',\ E',\ C',\ D',\ u').$$

The full set of these rules is given in fig. 9, again in the order in which they need to be matched against machine states. To facilitate comparison with the state transition rules of the weakly normalizing counterpart as given in fig. 2, the same enumeration of rules has been chosen. The rules that complement existing rules

[11] Note that the entire *apps* $-$ $-$*lambs* corner in between disappears due to the β-reduction-in-the-large that effects the selection.

Returning from β-reductions with closures on S
$(7b)\ \ ([\,E_b\,\Lambda\,e_b\,] : S,\ E,\ nil,\ (E',\ C',\ D',\ u'),\ u)\ \to ([\,E_b\,\Lambda\,e_b\,] : S,\ E',\ C',\ D',\ u')$

Rearranging applications on C
$(1)\ \ (S,\ E,\ @\,e_f\ e_a : C,\ D,\ u)\ \to ([\,E\ e_a\,] : S,\ E,\ e_f : @ : C,\ D,\ u)$

Creating closures on S for abstractions on C
$(2)\ \ (S,\ E,\ \Lambda\,e_b : C,\ D,\ u)\ \to ([\,E\ \Lambda\,e_b\,] : S,\ E,\ C,\ D,\ u)$

Substituting deBruijn indices
$(3)\ \ (S,\ E,\ \ \#i : C,\ D,\ u)\ \to (lookup(\,\#i,\ u,\ E\,) : S,\ E,\ C,\ D,\ u)$

Entering the evaluation of β-redices
$(4a)\ \ ([\,E'\ \Lambda\,e_b\,] : e_a : S,\ E,\ @ : C,\ D,\ u)\ \to (S,\ e_a : E',\ e_b : nil,\ (E,\ C,\ D,\ u),\ u)$

Dealing with unapplied closures on S
$(4b)\ \ ([\,E'\ \Lambda\,e_b\,] : S,\ E,\ C,\ D,\ u)\ \to (S,\ (u+1) : E',\ e_b : \Lambda : nil,\ (E,\ C,\ D,\ u),\ (u+1))$

Entering the normalization of suspensions on S
$(5)\ \ ([\,E'\ e'\,] : S,\ E,\ C,\ D,\ u)\ \to (S,\ E',\ e' : nil,\ (E,\ C,\ D,\ u),\ u)$

Putting leading Λs in front of an expression in S
$(4c)\ \ (e_b : S,\ E,\ \Lambda : nil,\ (E',\ C',\ D',\ u'),\ u)\ \ \to (\Lambda\,e_b : S,\ E',\ C',\ D',\ u')$

Dealing with abstractions on S and apply nodes on C
$(8)\ \ (\Lambda\,e_b : S,\ E,\ @ : C,\ D,\ u)\ \to (S,\ E,\ \Lambda\,e_b : @ : C,\ D,\ u)$

Rearranging applications for the evaluation of tail suspensions
$(9)\ \ (e_b : [\,E'\ e_a\,] : S,\ E,\ @ : C,\ D,\ u)\ \to ([\,E'\ e_a\,] : e_b : S,\ E,\ @^* : C,\ D,\ u)$

Reconstructing applications after normalization of their tail suspensions
$(10)\ \ (e_a : e_b : S,\ E,\ @^* : C,\ D,\ u)\ \to (@\,e_b\,e_a : S,\ E,\ C,\ D\ u)$

Reconstructing irreducible applications in S
$(6)\ \ (\ e_b : e_a : S,\ E,\ @ : C,\ D,\ u)\ \to (@\ e_b\ e_a : S,\ E,\ C,\ D,\ u)$

Returning from β-reductions and η-extensions
$(7a)\ \ (S,\ E,\ nil,\ (E',\ C',\ D',\ u'\ )\ u\ ) \to (S,\ E',\ C',\ D',\ u'\ )$

Fig. 9. The state transition rules of a fully normalizing FN_SECD machine

have their numbers tagged by letters $b,\ c$ (with a tagging the original rules), and three entirely new rules receive the numbers 8, 9 and 10.

Rules (1) to (4a), other than for an additional variable u that holds the current ULC value, are exactly the same as those of the weakly normalizing machine. The function *lookup* used in rule (3) is per pattern matching recursively defined as:

$$\begin{array}{rcl} lookup\,(\#0,\ u,\ [\,E'\ e'\,] : E) & \to & [\,E'\ e'\,] \\ (\#0,\ u,\ un : E) & \to & \#(u - un) \\ (\#i,\ \ u,\ v : E) & \to & lookup\,(\#(i-1),\ u,\ E) \end{array}$$

i.e., it returns as the i-th environment entry either a suspension or, if this entry contains a ULC value un, the corresponding deBruijn index.

Rule (4b) η-extends the unapplied abstraction contained in a closure that sits on top of stack S. It does so by prepending the current ULC, incremented by one, to the closure's environment that now becomes active, and by setting the isolated abstraction body up in C for evaluation. To complete the η-extension, the Λ is squeezed underneath the abstraction body, from where it may be retrieved once the body is completely evaluated. Also, the machine saves on the dump a return continuation that includes the old ULC, and it continues with the updated ULC in what now has become the current context. Rule (4c) intercepts the complementary stack configuration that has the evaluated abstraction body on top of S and a Λ as the sole entry on top of C. From these components it constructs a head-normalized abstraction on S. The return continuation retrieved from the dump also includes the old ULC value, which happens to be the current value decremented by one.

There are two rules that are complementary to those that save current machine states (or contexts) on the dump. Rule (7a) covers the general case of returning to a calling context whenever the code structure becomes empty, i.e., an instantiated abstraction body has been evaluated and in this form been completely moved from C to S. This rule must be called after all the other rules have failed to match. However, there is also the special case of an empty code structure in conjunction with a closure on top of S. Such configurations may come about when retrieving, by means of the function *lookup*, tail suspensions that happen to contain abstractions (and thus are in fact closures) from the environment. They must be caught before trying any of the other rules that expect closures on top of S, specifically rule (4b); hence rule (7b) as the first of the list.

Of the new rules, rule (8) takes care of the special case that an abstraction may end up as value on top of stack S together with an apply node @ in C, relative to which it is in operator position. This rule simply moves the abstraction back to C so that rule (2) may, in preparation for an application of rule (4a), wrap it up in a closure that is returned to S.

The remaining new rules (9) and (10) are to force and return from (head-) normalizing tail suspensions left over in a head-normalized spine. To figure out what must be done here, we need to understand that a machine that is just head-normalizing would produce in the value stack S an expression of the general form

$$\underbrace{\Lambda \ldots \Lambda}_{n}\ \underbrace{@ \ldots @}_{r}\ \#i\ [\ E_1\ t_1\ ] \ldots [\ E_r\ t_r\ ]\ \ ,$$

i.e., from top to bottom we have a leading *lambs* sequence followed by an *apps* sequence followed by a deBruijn index bound by one of the leading Λs followed by a sequence of tail expressions wrapped up in suspensions. But before this terminal state is reached, we have a configuration with the sequence

$$\#i\ [\ E_1\ t_1\ ] \ldots [\ E_n\ t_r\ ] \quad \text{in } S$$

and with the sequence

$$\underbrace{@\ldots@}_{r}\ \underbrace{\Lambda\ldots\Lambda}_{n}\ ,$$

of which the first @ is on top of C, and the remaining @s and Λs are, as parts of recursively nested contexts, stacked up in the dump D.

From this configuration forward, without rules (9) and (10) all apply nodes would be moved from C to S, using r times rules (6) and (7a), and then the Λs would follow, using n times rules (4c) and again (7a), which in fact means that the head-normalized spine would be assembled in S from the bottom (the head symbol $\#i$) up to the topmost Λ, without doing anything to the tail suspensions.

To evaluate, on the way up, the tails as well, the machine must intercept stack configurations with an apply node on top of C, an expression value other than a suspension (closure) on top of S and a suspension underneath, and to force the evaluation of this suspension. The first such configuration encountered has the head index $\#i$ of the head-normalized spine on top of S; all other configurations have irreducible application on top of S.

Rule (9), upon encountering such configuration, switches the first and the second expression, thus bringing the tail suspension to the top of S, which in turn enables rule (5) to effect its evaluation. At the same time, the apply node on top of C is marked with the superscript * to keep note of the fact that operator and operand have been interchanged. Upon returning the value (normal form) of the suspension to the top of S, rule (10) simply takes the two expressions on S and the apply node on C to construct a syntactically complete (irreducible) application on top of S, with operator and operand in the right order again.

5.3 Head-Normalizing a Λ-Expression: An Example

To illustrate how the FN_SECD-machine goes about doing its job, fig. 10 shows a sequence of representative configurations that it steps through when reducing the Λ-expression

$$\Lambda\Lambda\,@\,@\,\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2\ \#1\ \#0$$

just to head-normal form. All configurations shown (except the last one) are those at which the machine arrives after having processed either successive η-extensions of unapplied abstractions or successive β-redices[12].

The first configuration shown at the top of the figure depicts the situation after having η-extended the leading two Λs. The ULC indices 2 and 1 are stacked up in the environment and the remaining expression is still in C, with one of the abstractors underneath, while the other one is saved on the dump as part of the outer of two nested return continuations. Next follows the configuration after having rearranged the two nested applications on top of C. It has suspensions for their arguments #0 and #1 stacked up in S, and the applicators squeezed underneath the remaining abstraction in C. The third configuration depicts

[12] To accommodate the relevant steps of this sequence on a single page, the initial configuration which has the ULC value initialized with 0, the entire expression set up in the code structure C and all other structures empty is omitted.

Rules 2 and 4b twice ⇓

$$\begin{array}{r}
nil \mid S \\
2 : 1 : nil \mid E \\
@\,@\,\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2\ \#1\ \#0 : \Lambda : nil \mid C \\
(1 : nil,\ \Lambda : nil,\ (nil, nil, nil, 0),\ 1) \mid D \\
2 \mid u
\end{array}$$

Rule 1 twice ⇓

$$\begin{array}{r}
[\,2 : 1 : nil\ \ \#1\,] : [\,2 : 1 : nil\ \ \#0\,] : nil \mid S \\
2 : 1 : nil \mid E \\
\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2 : @ : @ : \Lambda : nil \mid C \\
(1 : nil,\ \Lambda : nil,\ (nil, nil, nil, 0),\ 1) \mid D \\
2 \mid u
\end{array}$$

Rules 2 and 4a twice ⇓

$$\begin{array}{r}
nil \mid S \\
[\,2 : 1 : nil\ \ \#0\,] : [\,2 : 1 : nil\ \ \#1\,] : 2 : 1 : nil \mid E \\
\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2 : nil \mid C \\
(2 : 1 : nil,\ \Lambda : nil, (1 : nil,\ \Lambda : nil,\ (nil, nil, nil, 0),\ 1),\ 2) \mid D \\
2 \mid u
\end{array}$$

Rules 2 and 4b once ⇓

$$\begin{array}{r}
nil \mid S \\
3 : [\,2 : 1 : nil\ \ \#0\,] : [\,2 : 1 : nil\ \ \#1\,] : \#2 : \#1 : nil \mid E \\
@\,@\,\Lambda\,\#4\ \#3\ \#2 : \Lambda : nil \mid C \\
(2 : 1 : nil,\ \Lambda : nil, (1 : nil,\ \Lambda : nil,\ (nil, nil, nil, 0),\ 1),\ 2) \mid D \\
3 \mid u
\end{array}$$

Rule 1 twice ⇓

$$\begin{array}{r}
[\,E'\ \ \#3\,] : [\,E'\ \ \#2\,] : nil \mid S \\
E'\ =\ 3 : [\,2 : 1 : nil\ \ \#0\,] : [\,2 : 1 : nil\ \ \#1\,] : 2 : 1 : nil \mid E \\
\Lambda\,\#4 : @ : @ : \Lambda : nil \mid C \\
(2 : 1 : nil,\ \Lambda : nil, (1 : nil,\ \Lambda : nil,\ (nil, nil, nil, 0),\ 1),\ 2) \mid D \\
3 \mid u
\end{array}$$

Rules 2 and 4a once ⇓

$$\begin{array}{r}
[\,E'\ \ \#2\,] : nil \mid S \\
[\,E'\ \ \#3\,] : 3 : [\,2 : 1 : nil\ \ \#0\,] : [\,2 : 1 : nil\ \ \#1\,] : 2 : 1 : nil \mid E \\
\#4 : nil \mid C \\
(E',\ @ : \Lambda : nil, (2 : 1 : nil,\ \Lambda : nil, (1 : nil,\ \Lambda : nil,\ (nil, nil, nil, 0),\ 1),\ 2),\ 3) \mid D \\
3 \mid u
\end{array}$$

Rule 3 once ⇓

$$\begin{array}{r}
\#1 : [\,E'\ \ \#2\,] : nil \mid S \\
[\,E'\ \ \#3\,] : 3 : [\,2 : 1 : nil\ \ \#0\,] : [\,2 : 1 : nil\ \ \#1\,] : 2 : 1 : nil \mid E \\
nil \mid C \\
(E',\ @ : \Lambda : nil, (2 : 1 : nil,\ \Lambda : nil, (1 : nil,\ \Lambda : nil,\ (nil, nil, nil, 0),\ 1),\ 2),\ 3) \mid D \\
3 \mid u
\end{array}$$

Fig. 10. Snapshots of typical FN_SECD-machine configurations while head-normalizing the Λ-expression $\Lambda\Lambda\,@\,@\,\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2\ \#1\ \#0$. Note that all deBruijn indices are preceded by #, all *ULC*s are given as plain integers.

the situation after having completed the equivalent of two β-distributions: the argument suspensions are removed from S and prepended to the environment, while the applicators and abstractors involved are being consumed [13]. The following three configurations show the same steps being performed on the abstraction left in C, which return as the sole entry in C the index #4.

This configuration is conceptually equivalent to a 'straightened' spine consisting of a single $apps - lambs$-corner similar to the one in fig. 8. Here we have the original expression completely transformed into an environment whose entries are the tails of the apply nodes in the *apps*-sequence of such corner. Nothing except the head index of this expression is left in the code structure C.

Accessing the environment with this index, which corresponds to β-reducing in one conceptual step (or in-the-large) an equivalent $apps - lambs$-corner, picks the entry 2 which, after correction with the ULC value 3, is pushed into S as deBruijn index #1 (the last configuration).

At this point, the computation has in fact arrived at a head-normal form which, unfortunately, is not immediately obvious. Except for the updated head index in S and a tail suspension underneath, the constructor nodes of the head-normalized spine are recursively hidden in the dump. It contains four nestings of contexts that are being saved along the way, which include, from outermost to innermost, the code fragments $@ : \Lambda : nil$, $\Lambda : nil$, $\Lambda : nil$, nil. When appended to each other, they yield the trace $@ : \Lambda : \Lambda : \Lambda$. Prepending this trace in reverse order to the two entries in S would yield $\Lambda : \Lambda : \Lambda : @ : \#1 : [\ E' \ \#2\ ]$. Except for the separating symbols, this sequence equals the head-normal form of the initial expression.

However, since the machine doesn't stop there but computes full normal forms, it continues to first unsave, by means of rule (7a), the outermost context on D, thus restoring in C the code sequence $@ : \Lambda : nil$. This in turn enables rule (9) to force the evaluation of the tail suspension $[\ E' \ \#2\ ]$, which after several more steps returns the index value #2, and subsequently, after having recursively restored all the other contexts stacked up in the dump, the fully normalized expression $\Lambda\Lambda\Lambda\, @\, \#1\, \#2$.

6 The FN_SE(M)CD-Machine

In this section we introduce a more sophisticated version of the FN_SECD-machine that does away with some of the complications inherited from the original SECD-machine. Prime candidates for improvement are the state transition rules (2) and (4a/b) of the FN_SECD-machine (see fig. 9) which, irrespective of the contexts in which they are applicable, transform abstractions on top of C into closures on top of S, with the consequence that they have to be unwrapped again before reducing applications or η-extending unapplied abstractions, which almost always happens immediately afterwards. Moreover, as a closure on top of S may also

[13] Note that the environment that has built up in this configuration is in all subsequent steps abbreviated as E' to contain the representation of the dump structure in a single line.

originate from an environment access (rule (3)), it is imperative that the general approach be taken to create new contexts (and to save the current contexts in the dump) to ensure that the computation continues in the environment included in the closure. As an unpleasant side effect, head-normalizing only moderately long spines may generate deeply nested dump structures, as exemplified by the state transition sequence of fig. 10, since every single η-extension or β-distribution along a spine pushes another context (or return continuation). These contexts include, in nested form, successively growing environments and, as parts of the codes saved, the apply nodes and abstractors from which normalized spines must be (re)constructed. This is to say that the machine is predominantly busy saving on (and unsaving from) the dump increasingly complex structures that are pretty hard to analyze, e.g., when the machine is used in a step-by-step mode to follow up on some sequence of state transitions, say, for validation purposes.

The FN_SE(M)CD-machine avoids these problems by two fairly simple measures. It evaluates β-redices and η-extends unapplied abstractions directly, i.e., without going through the superfluous motions of creating closures, and thus avoids the excessive use of the dump. In fact, entire spines can be head-normalized in the same contexts, leaving the dump unchanged. New contexts need be created, and current contexts be saved, only when entering the evaluation of suspensions that are being retrieved from the environment, which happens either whenever a suspension is substituted into the head of a spine or whenever the tails of a head-normalized spine need to be normalized.

Moreover, the environments saved on the dump can be replaced by something much simpler. From the conceptual outline of head-order reduction in subsection 4.2 we recall that the $apps - lambs$-corner that builds up when η-extending and β-distributing cuts along a spine (compare figs. 7 and 8) is being consumed when β-reducing it in-the-large. This translates into the environment becoming irrelevant once it has been accessed by a head index which substitutes an environment entry in its place. If this entry happens to be a suspension, it creates a new context in which it is being evaluated, routinely saving a return continuation on the dump. This immediately raises the question of what the environment to be saved must look like now that the one that is part of the calling context has become obsolote. Another problem relates to the questions of what needs to be done about deBruijn indices that belong to the (head-) normalized expression returned after evaluating the suspension, and which role is being played by the ULCs in this new setting.

To find answers to these questions, we simply need to have a closer look at the spine of fig. 8. Once β-reduction-in-the-large has eaten up the entire $apps - lambs$-corner, there is basically only a leading sequence of η-extended Λs left of the original spine. As a head index bound by one of these leading Λs must, upon returning to the calling context, find an environment entry for it, all that needs to be done conceptually is to η-extend this leading $lambs$-sequence once more, which generates an $apps$-sequence of equal length that has in its tails ULCs in ascending order. And this is exactly the environment that must be included in a return continuation.

So, the solution to our problems consists in replacing in our machine state description the plain ULC variable u by a stack U which, beginning with the initial value 0, stacks up ULCs in their order of creation. This stack is made part of the context stored in the dump. Whenever a context is retrieved from the dump, the contents of this stack become the new (initial) environment. The current ULC value that is required to compute correct deBruijn indices is the topmost entry of U.

The machine also employs a special shunting yard mechanism that uses a separate trace stack M to temporarily store the sequence of abstractors and applicators encountered while traversing a spine from the root node down to the current position of activity. In any state of program execution, it contains the Λs that belong to the leading *lambs*-sequence that has built up at that point, and on top of it apply nodes @ that may or may not be consumed by further β-reductions.

The FN_SE(M)CD-machine derives from the weakly normalizing SE(M)CD-machine described in [Kge05] whose specification, unfortunately, is erroneous, but under `www.informatik.uni-kiel.de/inf/Kluge/index-de.html` a corrected version may be found. Relative to the FN_SECD-machine, its specification requires a few more state transition rules as more stack configurations need to be distinguished, specifically with regard to the topmost entries on the trace stack M. The rules are generally simpler and more direct, operating just locally on the stack tops. Except for environment accesses, there is no need to digg deeper than two entries into a stack.

6.1 Traversing the Spine

Traversing the spine of an expression in the FN_SE(M)CD-machine involves just the code structure C, the value stack S and the trace stack M. They are operated like a shunting yard to traverse constructor expressions in pre-order. The expressions are initially set up in pre-order linearized form in C and from there moved to S. To preserve pre-order linearization in S, the constructor symbols Λ and @ are temporarily sidelined in M while their subexpressions, following recursively the same mechanism, are moved from C to S, where they end up with their left and right subexpressions interchanged [Ber75].

To describe how this traversal mechanism works, we consider a very basic machine only whose state is given by a triple $(S,\ M,\ C)$. The expressions to be traversed are assumed to have the general form $\kappa\, e_1\, e_2\, \ldots\, e_n$, where κ is an n-ary constructor and $e_1, \ldots, e_n$ are subexpressions; they are in C set up for traversal as sequences $\kappa : e_1 : e_2 : \ldots : e_n : C$.

The state transition rules of this machine are given below, listed in the order in which they must be matched against machine states:

$$(S,\ \kappa^{(0)} : nil,\ C) \rightarrow (\kappa : S,\ nil,\ C)$$

$$(S,\ \kappa_1^{(0)} : \kappa_2^{(i)} : M,\ C) \mid i > 0 \rightarrow (\kappa_1 : S,\ \kappa_2^{(i-1)} : M,\ C)$$

$$(S,\ \kappa^{(i)} : M,\ e_{at} : C) \mid i > 0 \rightarrow (e_{at} : S,\ \kappa^{(i-1)} : M,\ C)$$

$$(S,\ M,\ \kappa : C) \rightarrow (S,\ \kappa^{(n)} : M,\ C)$$

The last rule moves a constructor symbol that appears on top of C into the trace stack M and attaches to it an index which initially receives as value the arity n. This index denotes the number of subexpressions hooked up to the constructor that are still lined up in C. The third rule specifies how the index is decremented upon moving an atomic subexpression e_{at} from C to S.

Completing the traversal of a constructor expression is captured by the first two rules. A constructor with arity 0 on top of M indicates that all its subexpressions have been moved to S, i.e., none are left in C, and the traversal of the entire expression can be completed by moving the constructor from M to S. The two rules distinguish between the trace stack underneath being empty and another constructor being underneath, in which case its index must be decremented to notify completion in S of one of its subexpressions. Discriminating between these two trace stack configurations is required in several of the state transition rules of the full FN_SE(M)CD-machine.

The machine terminates with both M and C being empty as there is no rule with which to continue.

The sequence of state transitions shown in fig. 11 illustrates how this machine traverses the expression $\kappa\, a\, \kappa\, b\, c$, where κ is assumed to be a binary constructor,

$$\begin{tikzcd}
\begin{array}{r} \kappa : a : \kappa : b : c : nil \mid C \\ nil \mid M \\ nil \mid S \end{array} \arrow[r, Rightarrow] & \begin{array}{r} a : \kappa : b : c : nil \mid C \\ \kappa^{(2)} : nil \mid M \\ nil \mid S \end{array} \arrow[d, Rightarrow] \\
\begin{array}{r} b : c : nil \mid C \\ \kappa^{(2)} : \kappa^{(1)} : nil \mid M \\ a : nil \mid S \end{array} \arrow[d, Rightarrow] & \begin{array}{r} \kappa : b : c : nil \mid C \\ \kappa^{(1)} : nil \mid M \\ a : nil \mid S \end{array} \arrow[l, Rightarrow] \\
\begin{array}{r} c : nil \mid C \\ \kappa^{(1)} : \kappa^{(1)} : nil \mid M \\ b : a : nil \mid S \end{array} \arrow[r, Rightarrow] & \begin{array}{r} nil \mid C \\ \kappa^{(0)} : \kappa^{(1)} : nil \mid M \\ c : b : a : nil \mid S \end{array} \arrow[d, Rightarrow] \\
\begin{array}{r} nil \mid C \\ nil \mid M \\ \kappa : \kappa : c : b : a : nil \mid S \end{array} & \begin{array}{r} nil \mid C \\ \kappa^{(0)} : nil \mid M \\ \kappa : c : b : a : nil \mid S \end{array} \arrow[l, Rightarrow]
\end{tikzcd}$$

Fig. 11. Traversing the expression $\kappa\ a\ \kappa\ b\ c$ from stack C to S via stack M

as for instance the applicator @, and a, b, c are atomic expressions. The traversal begins with the stack configuration in the upper left, which has the expression set up in C (with M and S being empty), and continues along the double arrows until it terminates with the stack configuration in the lower left, which has the expression in left-right transposed pre-order linearized form reconstructed in S; M and C are empty.

It is interesting to note that this mechanism recursively brings about configurations in which the components of binary constructor expressions are spread out over the tops of the stacks involved, i.e., with a constructor whose index is 1 on top of M, its right subexpression on top of C and its left subexpression on top of S (the third and fifth configurations). If the expressions would be more complex, e.g., applications of abstractions, then such configurations could be easily intercepted to perform β-reductions by popping their components off the tops of the stacks and pushing into S their values instead.

Using stack M as a temporary storage for constructor symbols is the key to performing almost all state transformations in the FN_SE(M)CD-machine as local operations that involve just stack tops, which are fairly straightforward to implement in a real machine.

6.2 The State Transition Rules

A state of the full FN_SE(M)CD-machine is described by a six-tuple

$$(S,\ E,\ M,\ C,\ D,\ U)\ \ ,$$

which, other than including the trace stack M and replacing the ULC variable u with the stack U, is the same as that of the FN_SECD-machine of section 5. The structures saved on (and unsaved from) the dump accordingly change to $(U,\ M,\ C,\ D)$.

The applicators @ or the abstractors Λ that appear on top of the trace stack, in conjunction with the indices attached to them, play a decisive role in almost all state transition rules. As before, these rules are in the order in which they need to be checked against machine states given in fig. 12[14].

Rule (1) prepares in one step applications for evaluation. It does so by creating in S a suspension for the operand expression e_a and by pushing the applicator with index 1 into the trace stack M, indicating that only its operator expression remains in C. This transformation, which in fact interchanges the positions of operator and operand relative to the applicator, can be split up into the following sequence of simpler state transitions

$$(1a)\ (S,\ E,\ M,\ @\,e_f\,e_a : C,\ D,\ U)\ \rightarrow\ (S,\ E,\ M,\ @ : e_a : e_f : C,\ D,\ U)$$

$$(1b)\ (S,\ E,\ M,\ @ : e_a : e_f : C,\ D,\ U)\ \rightarrow\ (S,\ E,\ @^{(2)} : M,\ e_a : e_f : C,\ D,\ u)$$

$$(1c)\ (S,\ E,\ @^{(2)} : M,\ e_a : e_f : C,\ D,\ U)\ \rightarrow\ ([\,E\ e_a\,] : S,\ E,\ @^{(1)} : M,\ e_f : C,\ D,\ U)$$

[14] The constructor κ that is used in rules (4b) and (5b) stands for either @ or Λ.

which, after having flipped operator and operand, follows more closely the elementary traversal steps specified in the preceding subsection.

Rule (2) applies abstractions directly to operands in S (which due to prior application of rule (1) are bound to be suspensions). Rule (3) has unapplied abstractions prepend current ULC values (incremented by one) to the active environment; it also pushes the ULC into stack U and the abstractor into the trace stack. Rules (4a/b) effect environment accesses for deBruijn indices occuring on top of the code structure C. Since pushing into S an environment entry

Returning from β-reductions_in_the_large
(0) $(S,\ E,\ nil,\ nil,\ (U',\ M',\ C',\ D'),\ U)\ \rightarrow\ (S,\ U',\ M',\ C',\ D',\ U')$

Spreading applications on top of C out over C, M, S
(1) $(S,\ E,\ M,\ @\,e_f\,e_a : C,\ D,\ U)\ \rightarrow\ ([\,E\ e_a\,] : S,\ E,\ @^{(1)} : M,\ e_f : C,\ D,\ U)$

Applying abstractions on top of C to operands on top of S
(2) $(e_a : S,\ E,\ @^{(1)} : M,\ \Lambda e_b : C,\ D,\ U)\ \rightarrow\ (S,\ e_a : E,\ M,\ e_b : C,\ D,\ U)$

η-extending unapplied abstractions on top of C
(3) $(S,\ E,\ M,\ \Lambda e_b : C,\ D,\ u : U)\ \rightarrow\ (S,\ (u+1) : E,\ \Lambda^{(1)} : M,\ e_b : C,\ D,\ (u+1) : u : U)$

Substituting deBruijn indices by environment entries
(4a) $(S,\ E,\ nil,\ \#i : C,\ D,\ u : U)\ \rightarrow\ (lookup(\,\#i,\ u,\ E\,) : S,\ E,\ nil,\ C,\ D,\ u : U)$

(4b) $(S,\ E,\ \kappa^{(j)} : M,\ \#i : C,\ D,\ u : U) \mid (j > 0)\ \rightarrow\ (lookup(\,\#i,\ u,\ E\,) : S,\ E,\ \kappa^{(j-1)} : M,\ C,\ D,\ u : U)$

Applying closures on top of S to operands underneath
(5a) $([\,E'\ \Lambda e_b\,] : e_a : S,\ E,\ @^{(0)} : nil,\ C,\ D,\ U)\ \rightarrow\ (S,\ e_a : E',\ nil,\ e_b : nil,\ (U,\ nil,\ C,\ D),\ U)$

(5b) $([\,E'\ \Lambda e_b\,] : e_a : S,\ E,\ @^{(0)} : \kappa^{(j)} : M,\ C,\ D,\ U) \mid (j > 0)\ \rightarrow$
$(S,\ e_a : E',\ nil,\ e_b : nil,\ (U,\ \kappa^{(j-1)} : M,\ C,\ D),\ U)$

Setting suspensions on top of S up for evaluation
(6) $([\,E'\ e_a\,] : S,\ E,\ M,\ C,\ D,\ U)\ \rightarrow\ (S,\ E',\ nil,\ e_a : nil,\ (U,\ M,\ C,\ D),\ U)$

Constructing abstractions on top of S from their components on M and C
(7a) $(e_b : S,\ E,\ \Lambda^{(0)} : nil,\ nil,\ D,\ u : U)\ \rightarrow\ (\Lambda e_b : S,\ E,\ nil,\ nil,\ D,\ U)$

(7b) $(e_b : S,\ E,\ \Lambda^{(0)} : \Lambda^{(1)} : M,\ nil,\ D,\ u : U)\ \rightarrow\ (\Lambda e_b : S,\ E,\ \Lambda^{(0)} : M,\ nil,\ D,\ U)$

Moving abstractions from S to C
(8) $(\Lambda e_b : S,\ E,\ @^{(0)} : M,\ C,\ D,\ U)\ \rightarrow\ (S,\ E,\ @^{(1)} : M,\ \Lambda e_b : C,\ D,\ U)$

Setting tail suspensions up for evaluation
(9) $(e_b : [\,E'\ e'_a\,] : S,\ E,\ @^{(0)} : M,\ C,\ D,\ U)\ \rightarrow\ (S,\ E',\ nil,\ e'_a : nil,\ (U,\ @^{*} : M,\ e_b : C,\ D),\ U)$

Returning from evaluating tail suspensions
(10) $(e_a : S,\ E,\ @^{*} : M,\ e_b : C,\ D,\ U) \rightarrow\ (e_b : e_a : S,\ E,\ @^{(0)} : M,\ C,\ D,\ U)$

Reconstructing irreducible applications on S
(11a) $(e_b : e_a : S,\ E,\ @^{(0)} : nil,\ C,\ D,\ U)\ \rightarrow\ (@\,e_b\,e_a : S,\ E,\ nil,\ C,\ D,\ U)$

(11b) $(e_b : e_a : S,\ E,\ @^{(0)} : \kappa^{(j)} : M,\ C,\ D,\ U) \mid (j > 0)\ \rightarrow\ (@\,e_b\,e_a : S,\ E,\ \kappa^{(j-1)} : M,\ C,\ D,\ U)$

Fig. 12. The state transition rules of a fully normalizing SE(M)CD-machine

accessed by a deBruijn index found on C is in fact equivalent to moving this entry from C to S, rule (4b) must also decrement the index attached to the constructor symbol on top of M to signal completion of the traversal of one of its subexpressions.

Environment accesses that return on S suspensions which are substituted for deBruijn indices in head (or operator) positions are taken care of by rules (5a/b) and (6) [15]. The special case that such a suspension is in fact a closure, i.e., contains an abstraction, effects creation of a new context in which the abstraction body is set up for evaluation in the environment that comes with the closure. This environment is prepended by the operand of the particular application that in S is found underneath the closure. The general case of a suspension on top of S leads to the creation of a new context in which it is evaluated. The old context is in either case saved on the dump.

Suspensions in the tails of apply nodes that are left over in a head-normalized spine are by rule (9) set up for evaluation in new contexts. Being in the tail of an application whose head (operator) is already evaluated is identified as being the second-to-top entry in the value stack S relative to an applicator $@^{(0)}$ on top of M. Entering and returning from evaluating a tail suspension necessitates a reverse traversal step which moves the operator expression back to the control structure C in order to bring the suspension to the top of S prior to creating a new context. This can be made more explicit by splitting rule (9) up in two consecutive steps:

$$(9a)\ (e_b : [\, E'\ e'_a\,] : S,\ E,\ @^{(0)} : M,\ C,\ D,\ U)\ \rightarrow\ ([\, E'\ e'_a\,] : S,\ E,\ @^* : M,\ e_b : C,\ D,\ U)$$

$$(9b)\ ([\, E'\ e'_a\,] : S,\ E,\ @^* : M,\ e_b : C,\ D,\ U)\ \rightarrow\ (S,\ E',\ nil,\ e'_a : nil,\ (U,\ @^* : M,\ e_b : C,\ D),\ U)$$

The special applicator $@^*$ serves as a marker that signifies evaluation of its tail suspension. Upon returning from the new context, rule (10) uses this applicator to restore the original stack configuration with the suspension, now evaluated to e_a, underneath e_b on S and the applicator $@^{(0)}$ again on top of M.

Returning from evaluating a suspension in another context is accomplished by rule (0). It intercepts a configuration with an empty control structure and an empty trace stack, which signals completion of traversing, and thereby evaluating, an expression from C to S, and restores the calling context saved as return continuation on the dump D. Note that the contents of stack U that are included in the return continuation becomes the new environment.

The remaining rules (7a/b) and (11a/b) are to construct in S from the @s and Λs that have accumulated in the trace stack the *apps* and *lambs* sequences, respectively, of a head-normalized spine.

6.3 Head-Normalizing the Expression of Subsection 5.3

Fig. 13 illustrates, again as a sequence of stack configurations, how the FN_SE(M) CD-machine reduces the spine of a Λ-expression to head-normal form. To expose

[15] Note that a suspension on top of S being in head (operator) position of an application is identified by the index 0 attached to the applicator that sits on top of M.

Rule 3 twice $\Downarrow$

$$\begin{array}{r}
nil \mid S \\
2 : 1 : nil \mid E \\
\Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
@\,@\,\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2\ \#1\ \#0 : nil \mid C \\
nil \mid D \\
2 : 1 : 0 : nil \mid U
\end{array}$$

Rule 1 twice $\Downarrow$

$$\begin{array}{r}
[\,2 : 1 : nil\ \ \#1\,] : [\,2 : 1 : nil\ \ \#0\,] : nil \mid S \\
2 : 1 : nil \mid E \\
@^{(1)} : @^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2 : nil \mid C \\
nil \mid D \\
2 : 1 : 0 : nil \mid U
\end{array}$$

Rule 2 twice $\Downarrow$

$$\begin{array}{r}
nil \mid S \\
[\,2 : 1 : nil\ \ \#0\,] : [\,2 : 1 : nil\ \ \#1\,] : 2 : 1 : nil \mid E \\
\Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2 : nil \mid C \\
nil \mid D \\
2 : 1 : 0 : nil \mid U
\end{array}$$

Rule 3 once $\Downarrow$

$$\begin{array}{r}
nil \mid S \\
3 : [\,2 : 1 : nil\ \ \#0\,] : [\,2 : 1 : nil\ \ \#1\,] : 2 : 1 : nil \mid E \\
\Lambda^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
@\,@\,\Lambda\,\#4\ \#3\ \#2 : \Lambda : nil \mid C \\
nil \mid D \\
3 : 2 : 1 : 0 : nil \mid U
\end{array}$$

Rule 1 twice $\Downarrow$

$$\begin{array}{r}
[\,E'\ \ \#3\,] : [\,E'\ \ \#2\,] : nil \mid S \\
E' \ = \ 3 : [\,2 : 1 : nil\ \ \#0\,] : [\,2 : 1 : nil\ \ \#1\,] : 2 : 1 : nil \mid E \\
@^{(1)} : @^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
\Lambda\,\#4 : nil \mid C \\
nil \mid D \\
3 : 2 : 1 : 0 : nil \mid U
\end{array}$$

Rule 2 once $\Downarrow$

$$\begin{array}{r}
[\,E'\ \ \#2\,] : nil \mid S \\
[\,E'\ \ \#3\,] : 3 : [\,2 : 1 : nil\ \ \#0\,] : [\,2 : 1 : nil\ \ \#1\,] : 2 : 1 : nil \mid E \\
@^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
\#4 : nil \mid C \\
nil \mid D \\
3 : 2 : 1 : 0 : nil \mid U
\end{array}$$

Rule 4b once $\Downarrow$

$$\begin{array}{r}
\#1 : [\,E'\ \ \#2\,] : nil \mid S \\
[\,E'\ \ \#3\,] : 3 : [\,2 : 1 : nil\ \ \#0\,] : [\,2 : 1 : nil\ \ \#1\,] : 2 : 1 : nil \mid E \\
@^{(0)} : \Lambda^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
nil \mid C \\
nil \mid D \\
3 : 2 : 1 : 0 : nil \mid U
\end{array}$$

Fig. 13. Snapshots of typical FN_SE(M)CD-machine configurations while head-normalizing the Λ-expression $\Lambda\Lambda\,@\,@\,\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2\ \#1\ \#0$

the differences relative to the FN_SECD-machine of the preceding section, we have chosen as an example again the expression

$$\Lambda\Lambda\,@\,@\,\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2\ \#1\ \#0$$

as in fig. 10 in order to be able to compare on a step-by-step basis how both machines are working.

The main difference that immediately catches the eye is that the dump structure D of the FN_SE(M)CD remains empty throughout the entire head-normalizing sequence since none of the η-extensions or β-distributions creates a new context. This has been made possible by eliminating the superfluous steps of creating closures for abstractions that can be directly applied or η-extended, as a consequence of which the machine also accomplishes with one rule application each the same as the FN_SECD-machine with two.

The applicators and abstractors encountered along the spine, minus the @s that are being consumed by β-reductions/distributions, are now building up in the trace stack M, cleanly separated from the code structure that must accommodate them in the FN_SECD-machine. In the head-normalized configuration at the bottom, this trace includes in the form of a flat sequence exactly those constructor symbols from which the spine of the full normal form must be assembled.

This sequence of machine configuration also shows how the stack U up to the point of head-normalization grows to three non-zero entries pushed by the three unapplied Λs along the spine.

Most importantly, it may be noted that the items stacked up in the value stack S and in the environment structure E are in each of the configurations exactly the same as in the equivalent configurations of the FN_SECD-machine (compare fig. 10), just as it should be.

Going through essentially the same sequences of state transformations when head-normalizing the same expression renders it reasonable to assume that both machines also compute the same normal forms. This may be concluded from the fact that reducing left-over tail suspensions of head-normalized spines is nothing but recursively applying head-normalization to the spines of the respective expressions in their own environments.

7 A Fully Normalizing Graph Reducer

We will now briefly outlined a conceivable implementation of the FN_SE(M)CD-machine as a graph reducer [16].

Graph reduction is the standard implementation technique for functional languages, particularly for those with a lazy semantics [Joh84, PeyJ92, PvE93]. The idea is to represent Λ-expressions in a form that hides (sub)expressions of constructor symbols and also environment structures recursively behind pointers and to perform reductions primarily as pointer manipulations, which typically brings down from $O(n^2)$ to $O(n)$ the runtime complexity of programs that

[16] The contents of this section are in parts adopted from the authors monograph on Abstract Computing Machines [Kge05].

operate on data structures of size n. Another important advantage of graph reduction relates to sharing the evaluation of (sub)expressions among several points of substitution. This technique is the key to optimizing normal order reduction strategies with regard to numbers of reduction steps performed.

Under a head-order regime, opportunities for sharing primarily arise whenever suspensions are copied from the environment into head positions of spines. Reducing them in these places may then be shared with the environment and thus with all occurrences of pointers to the suspensions from elsewhere.

7.1 Graphs and Graph Reduction

Following its syntax, graph representations of expressions of the pure Λ-calculus may be composed of just two types of inner nodes, these being the constructors @ and Λ, and of deBruijn indices as the only type of leaf nodes. The nodes are connected by directed edges leading from root nodes (to root nodes) to leaf nodes. In a conceivable implementation, the inner nodes may be represented as cells in a memory section called the heap which, in addition to the node symbols themselves, also include pointers to subgraphs. The leaf node cells just contain deBruijn indices (or other atomic symbols such as constant values, primitive function symbols, etc. of an applied λ calculus). Entire *lambs* sequences of length n may be represented by single node cells of the form $[\ \Lambda,\ n,\ p_h\ ]$ (with p_h being the pointer to the subgraph that represents the abstraction body) as no other graph structures are branching off, and apply nodes by cells of the form $[\ @,\ p_h,\ p_t\ ]$ (with $p_h,\ p_t$ being the pointers to the head and tail subgraphs, respectively). Suspension nodes take the form $[\ sus,\ p_E,\ p_e\ ]$, where sus denotes the node type, p_E is the pointer to the environment, and p_e is the pointer to the tail expression.

For instance, the head form depicted in fig. 4 of section 4 thus translates into the graph shown in fig. 14 below. The tails that are abbreviated here by the symbols e_j, of course, feature similar graph structures of their own.

A graph reducer typically uses such graphs as static (nondestructible) structures, or alternatively equivalent static code, in order to be able to share their application to different sets of nonshareable operands. These graphs are traversed from top to bottom along their spines to build environments from $apps - lambs$ corners encountered and to construct, from the bottom up, new graph structures somewhere else in the heap.

The environment is composed of frames corresponding to $apps-lambs$-corners which in their order of creation are linked by pointers. A frame contains as many pointers to suspensions and as many ULC entries as there are apply nodes in the *apps*-sequence and unapplied Λs in the *lambs*-sequence, respectively, in the particular corner, so that the frame size equals the length of the *lambs*-sequence (or the arity of the abstraction) involved. Each frame is preceded by a header which includes the link pointer to the frame deeper down in the environment, the number of frame entries, and the ULC value that applies in the particular (sub)environment.

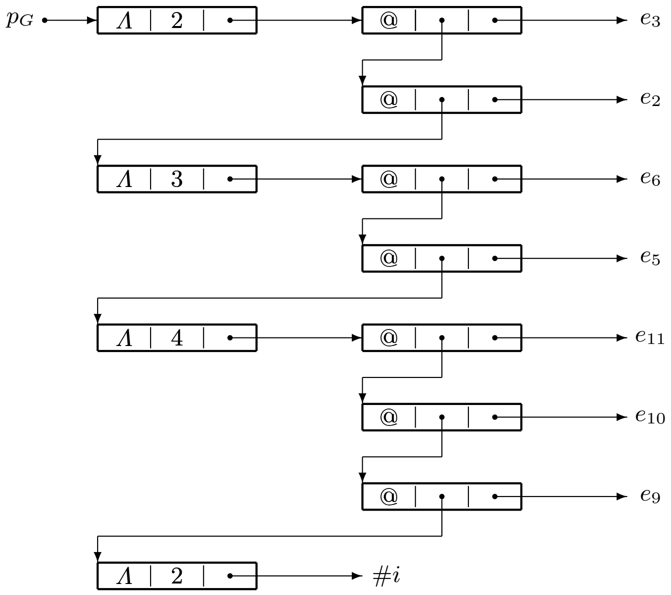


Fig. 14. Graph implementation of the spine of fig. 4

Organizing the environment as a structure of linked frames is due to the need to share, as a matter of efficiency, common parts among environments that belong to different contexts. A typical example is a tail suspension that comes with an environment E_1 contained in a larger environment E_2 of which the suspension is an entry. Substituting this suspension for a head index and evaluating it in this place means that new environment frames must be created on top of E_2 but linked up to the environment E_1 by a pointer that bridges the gap between the tops of E_2 and E_1.

As a minor inconvenience, accessing a particular environment entry with some index $\#i$ therefore entails conparing it with the size of the topmost frame and, if found larger, to subtract this size from the index and proceed down the link pointer to the next frame. This step must be repeated until the remaining index falls inside a frame.

The runtime structures necessary to perform graph reductions are the static graph (or equivalent static code), the environment, and a trace stack that serves a slightly different purpose than the one of the FN_SE(M)CD-machine. In fact, other than maintaining a pointer to the leading Λ-cell it assumes more or less the role of its value stack S: it stacks (and unstacks) pointers to the tail suspensions that are being created on the way down a spine and from which, after evaluating the suspensions left over in the trace stack following head normalization, the spine of a fully normalized graph (or subgraph) is reconstructed.

Reducing a graph sets out with an empty environment, a ULC set to 0, and a graph pointer p_G pointing to the topmost node cell of the graph. The pointer to this cell becomes the first entry of the trace stack.

The graph pointer then advances down the spine along the chain of head pointers p_h and pushes into the trace stack pointers to suspensions for tail expressions until the next Λ-cell is reached. This Λ-cell now controls the creation of an environment frame by filling it from the trace stack either with suspension pointers as the cell's arity parameter demands or until these pointers are prematurely exhausted, in which case the topmost Λ-cell (the pointer to which is at the bottom of the trace stack) pops to the top, indicating unapplied Λs. In this latter case, the remaining frame entries are filled with ULCs and the number of ULCs pushed is added to the arity of the Λ-cell in the trace stack, thus in fact completing an η-extension.

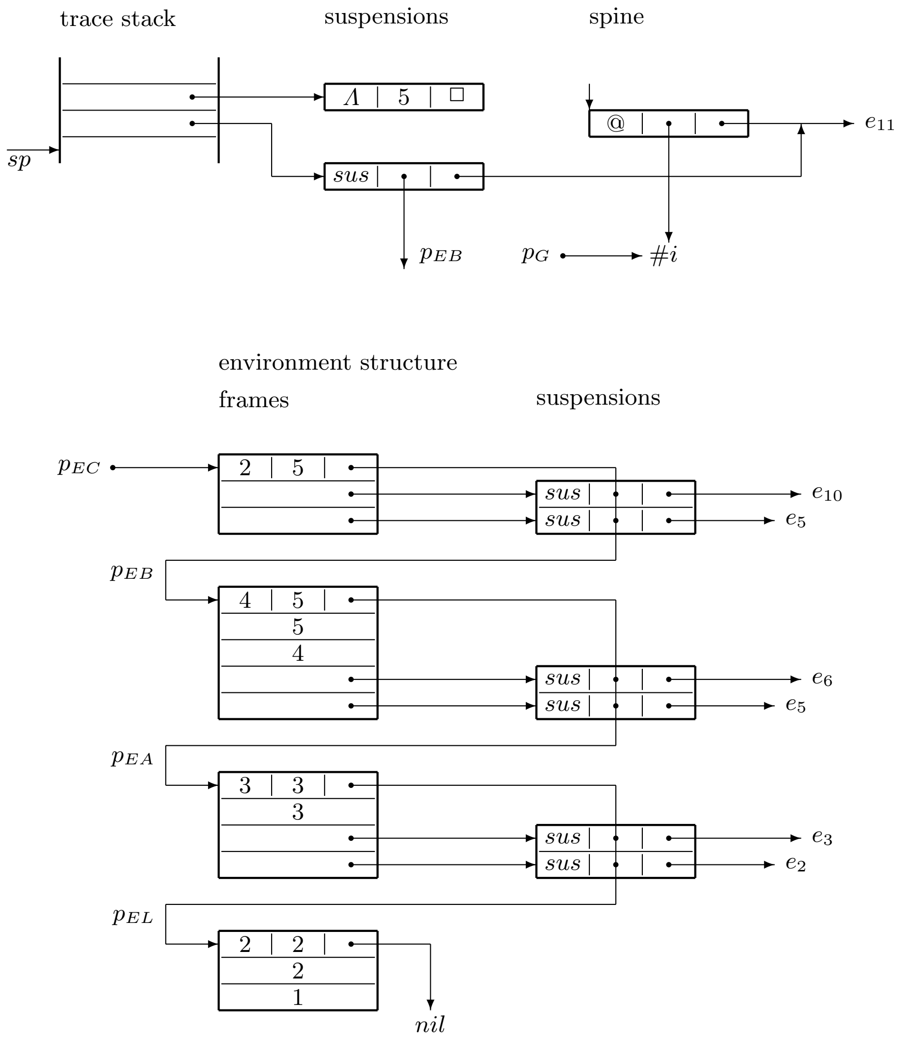


Fig. 15. Trace stack and environment after having reached the head of the spine

Fig. 15 above shows how the trace stack and the environment look like after evaluation has arrived at the head of the spine, as indicated by the position of the graph pointer p_G. The trace stack just has a pointer to the suspension for the tail e_{11} sitting on top of the pointer to the topmost Λ-cell which has its arity index updated to 5. The environment that has built up at this point is composed of four frames corresponding to the three $apps - lambs$ corners of the original spine and a fourth frame at the bottom that is due the η-extension of the leading $lambs$ sequence. Note that the environments pointed to by p_{EC}, p_{EB}, p_{EA} and p_{EL} correspond to cuts C, B, A and L, respectively, in the straightened head form of fig. 8.

7.2 Continuing with Reductions in the Head

If the head index to which p_G is pointing selects from the environment another suspension, then in a straightforward approach its graph would have to be substituted for this head index and head-order reduction would have to continue along the extended spine in the environment carried along with the suspension. This may be accomplished by setting the graph pointer p_G and the environment pointer p_E to those found in the suspension node, and by setting the ULC to that of the topmost frame of the environment that comes with the suspension. If the suspension thus substituted is, or reduces to, an abstraction, then it creates a new environment frame by either consuming suspensions already held in the trace stack or by filling it with ULC entries.

The problem with these old suspensions held in the trace stack is that they constitute specific instantiations of the abstraction that renders reductions further down the extended spine dependent on them, i.e., evaluation of the suspension cannot be shared.

Sharing requires that a suspension selected by the head index be first reduced in isolation and that then the suspension node be overwritten with the resulting graph so that it can be seen by all pointers directed at it from somewhere else. The pointer to this graph may then be substituted for the head index, and reduction may continue along its spine.

The question that remains to be answered is just how far should the suspension be reduced in isolation. The safe thing to do is to reduce it just to head-normal form. If one exists, then we have a chance to reach a full normal form for the entire expression as well, but not necessarily so. If the result of head-normalizing the suspension would be an abstraction and the machine, on its way up the spine, would continue evaluating tail suspensions, some of them might not terminate, i.e., the computation would get trapped in a 'black hole'. However, if evaluating the suspension would stop with a head-normalized abstraction, and this abstraction would be applied to tail suspensions left in the trace stack, there might be a chance that the non-terminating tails are thrown away and the computation could terminate with a full normal form.

Substituting a head-normalized suspension for a head index is depicted in fig. 16. The upper part shows a typical trace stack and the lower end of a static graph with a deBruijn index in its head; the lower part shows the head-normalized spine of the suspension selected by the head index. The substitution

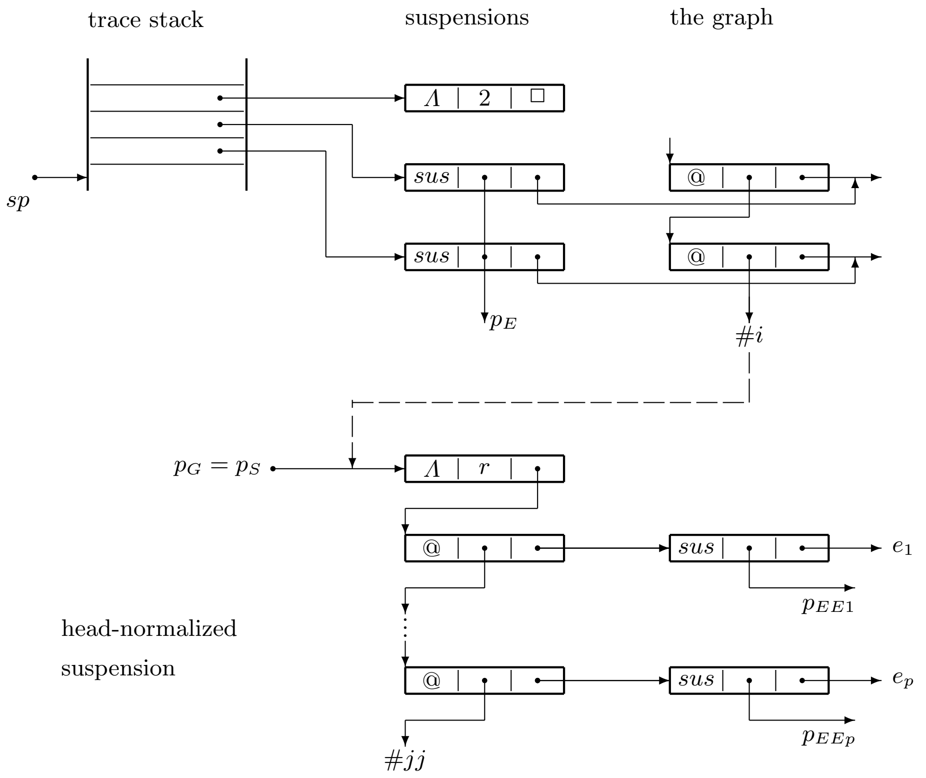


Fig. 16. Linking up to a head-normalized suspension in the head of a spine

is done simply by overwriting the graph pointer p_G with the pointer p_S to the topmost node of the new spine. The evaluation then continues by processing the *apps – lambs* corner formed by the two suspensions held in the trace stack and by the Λ-node to which p_G is now pointing.

Once the computation has arrived at a head-normal form, identified by a ULC value retrieved from the environment, the machine finally reverses gear, evaluates the suspensions left on the trace stack, and recursively constructs from the bottom up a spine of apply nodes (with a head index at the bottom) that are linked up by head pointers and whose tail pointers point to the normalized tail graphs. Finally, when arriving at the single Λ-node at the bottom of the trace stack, the spine is completed upwards by as many Λs as specified by the arity index in that node.

8 Related Work and Conclusion

Research on fully normalizing λ-calculus machines dates back to 1975 when Berkling proposed a string reduction machine whose most important operation was a complete and direct implementation of the β-reduction rule [Ber75].

To demonstrate the feasibility of the underlying concept, this proposal led to the construction of an experimental hardware machine that was completed in 1979 [Kge79]. To overcome the performance problems inherent in string reduction, Hommes published in 1982 an early graph reduction version of this machine [Hom82]. Also to mention is Wadsworth's work on a λ-calculus-based graph reducer [Wads71] which represents binding structures by pointers and performs β-reductions by pointer rearrangements.

In 1986 Berkling came up with the more sophisticated head-order reduction concept outlined in section 4 that describes environments completely within the framework of the nameles Λ-calculus. It employs η-extensions-in-the-large to fill in binding indices for missing arguments of abstractions and β-distributions-in-the-large to distribute environments over the components of applications [Ber86]. An alternative theoretical approach which achieves the same ends by slightly different means is the $\lambda\sigma$-calculus of Abadi, Cardelli, Curien and Levi [ACCL90]. It introduces environments through the notion of substitutions as an extension of the nameless Λ-calculus. Full normalization can be achieved by weakly normalizing machinery which must call upon a mechanism that pushes substitutions across unapplied abstractors and updates binding indices accordingly, which is equivalent to η-extension.

Based on Berklings work, Troullinos gives in his PhD thesis a formal specification of an abstract head-order reduction machine that, by a skillful choice of state transition rules, completely avoids a dump, using instead a direction parameter to distinguish between going up or down a spine [Trou93]. It also introduces the notion of an unapplied lambdas count as used by the SECD-machine descendants described in this paper. Crégut describes a fully normalizing abstract machine based on Krivine's weakly normalizing $\mathcal{K}$-machine [Kri85]. In the course of evaluating suspensions, it uses an updating scheme for binding indices that is similar to ULCs. Another fully normalizing machine is due to Grégoire and Leroy [GrLe02]. It augments a weakly normalizing machine by a so-called rollback function which, following the classical definition of the β-reduction rule, α-converts λ-bound variables in order to take them out of naming conflicts.

An early implementation of Berkling's head-order reduction concept is described in the PhD thesis by Hilton [Hil90]. An instruction-based fully-normalizing head-order reducer that makes extensive use of sharing reductions in the head is described in [Kge05]. This B-machine has been reconstructed from various unpublished drafts and handwritten notes by Berkling [Ber96, Ber97], and from the PhD theses by Hilton [Hil90] and Troullinos [Trou93].

Another instruction-based machine described in this monograph and earlier published in [GK96] employs weak normalization to do the routine work of naive substitutions when reducing full applications but switches to a special η-extension mechanism to deal with unapplied abstractions. Rather than filling in ULCs for missing arguments, it re-introduces the original variables, doing the equivalent of the transformation:

$$(\underbrace{\Lambda \dots \Lambda}_{n}\, e_0\, e_1 \dots e_k\ ) \mid k < n \ \to\ \lambda v_{k+1} \dots \lambda v_n((\underbrace{\Lambda \dots \Lambda}_{n}\, e_0\, e_1 \dots e_k\ )\ v_{k+1} \dots v_n).$$

The variables $\{\ v_{k+1} \ldots v_n\ \}$ are being retrieved from persistent structures in which have been saved the full parameter sets of all λ-abstractions of a program before converting them into equivalent nameless Λ-abstractions for reduction.

This η-extension mechanism makes use of the fact that unapplied abstractors always end up in a leading λ-sequence that never engages in further β-reductions but becomes part of the full normal form. Different instances of the same variables are distinguished by subscripts that enumerate the η-extension steps by which they have been introduced.

The two abstract machines described in this paper have been devised for the purpose of the summer school to convey the basic message that, starting from the well-known weakly normalizing SECD-machine, fully normalizing machines can be had by supplementing them with a few more state transition rules that η-extend unapplied abstractions, thus elegantly getting by the trouble of implementing full-fledged β-reductions. The key to performing these η-extensions with very little overhead is the use of ULC-indices rather than binding indices. It takes very simple updates to turn ULCs retrieved from the environment into deBruijn indices, in which form they must show up in (head-)normalized Λ-expressions.

Both the FN_SECD-machine and the FN_SE(M)CD-machine have been tested with the same set of about 25 example λ-expressions, all of which include in various contexts unapplied abstractions that require η-extension. Both machines have been found to reduce these expressions correctly to full normal forms. Of course, this is no proof that they work correctly for all Λ-expressions but it raises the level of confidence considerably.

References

[ACCL90] Abadi, M., Cardelli, L., Curien, P.-L., Levi, J.-J.: Explicit Substitutions. In: Proceedings of the 17th ACM Symposium on Principles of Programming Languages, pp. 1–16. ACM Press, New York (1990)

[Bar84] Barendregt, H.P.: The Lambda Calculus, Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam (1984)

[Ber75] Berkling, K.J.: Reduction Languages for Reduction Machines. In: Berkling, K.J. (ed.) Proceedings of the 2nd Annual Symposium on Computer Architecture, pp. 133–140. ACM/IEEE (1975)

[Ber86] Berkling, K.J.: Head-Order Reduction: a Graph Reduction Scheme for the Operational Lambda Calculus. LNCS, vol. 254, pp. 26–48. Springer, Heidelberg (1986)

[Ber96] Berkling, K.J.: The von Neumann-PLUS Architecture: an Evolutionary Development. unpublished draft (1996)

[Ber97] Berkling, K.J.: Privately communicated handwritten notes, Syracuse, NY (Spring 1997)

[Bird98] Bird, R.S.: Introduction to Functional Programming with Haskell, 2nd edn. Prentice-Hall, Englewood Cliffs (1998)

[Bru72] de Bruijn, N.G.: A Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church–Rosser Theorem. Indagationes Mathematicae 34, 381–392 (1972)

[CMQ83] Cardelli, L., McQueen, D.: The Functional Abstract Machine,The ML/LCF/HOPE Newsletter. AT&T Bell Labs, Murray Hill, NJ (1983)

[Chu41] Church, A.: The Calculi of Lambda Conversion. Princeton University Press, Princeton (1941)

[CCM85/87] Cousineau, G., Curien, P.L., Mauny, M.: The Categorial Abstract Machine. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 50–64. Springer, Heidelberg (1985); Science of Computer Programming, No. 8, 173–202 (1987)

[Cre90] Crégut, P.: An Abstract Machine for the Normalization of λ-Terms. In: Proceedings of the ACM Conference on LISP and Functional Programming, pp. 333–340 (1990)

[Dyb87] Dybvig, R.K.: The SCHEME Programming Language. Prentice-Hall, Englewood Cliffs (1987)

[GK96] Gaertner, D., Kluge, W.E.: π-RED^{+} – an Interactive Compiling Graph Reduction System for an Applied λ-Calculus. Journal of Functional Programming 6(5), 723–757 (1996)

[GrLe02] Grégoire, B., Leroy, X.: A Compiled Implementation of Strong Reduction. In: Proceedings of the ACM International Conference on Functional Programming, pp. 235–246 (2002)

[Hil90] Hilton, M.L.: Implementation of Declarative Languages, PhD thesis, CASE Center Technical Report No. 9008. Syracuse University, Syracuse, NY (1990)

[HS86] Hindley, J.R., Seldin, J.P.: Introduction to Combinators and λ-Calculus. London Mathematical Society Student Texts. Cambridge University Press, Cambridge (1986)

[Hom82] Hommes, F.: The Heap/Substitution Concept - an Implementation of Functional Operations on Data Structures for a Reduction Machine. In: Proc. 9th Annual Symposium on Computer Architecture, Austin (Texas), pp. 248–256 (1982)

[Joh84] Johnsson, T.: Efficient Compilation of Lazy Evaluation. ACM Conference on Compiler Construction, Montreal, Que., pp. 58–69 (1984)

[Kge79] Kluge, W.E.: The Architecture of the Reduction Machine Hardware Model, Internal Report GMD ISF-79-3, St. Augustin, Germany (1979)

[Kge05] Kluge, W.: Abstract Computing Machines – A Lambda Calculus Perspective. Springer, New York (2005)

[Kri85] Krivine, J.-L.: Un interpréte du λ-calcul. (1985)

[Lan64] Landin, P.J.: The Mechanical Evaluation of Expressions. The Computer Journal 6(4), 308–320 (1964)

[PeyJ92] Peyton Jones, S.L.: Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine. Journal of Functional Programming 2(2), 127–202 (1992)

[PvE93] Plasmeijer, R., van Eekelen, M.: Functional Programming and Parallel Graph Rewriting. Addison-Wesley, Reading (1993)

[Trou93] Troullinos, N.B.: Head-Order Techniques and Other Pragmatics of Lambda Calculus Graph Reduction, PhD thesis, CASE Center Technical Report No. 9322, Syracuse University, Syracuse, NY (1993)

[Ull98] Ullman, J.U.: Elements of MLProgramming, 2nd edn. Prentice-Hall, Englewood Cliffs (1998)

[Wads71] Wadsworth, C.P.: Semantics and Pragmatics of the Lambda Calculus, PhD thesis, Oxford University (1971)