An abstract machine for the normalization of λ-terms

P. Crégut
LIENS *
CNRS U.R.A. 1327

Abstract

Two abstract machines reducing terms to their full normal form are presented in this paper. They are based on Krivine's abstract machine [Kri85] which uses an environment to store arguments of function calls. A proof of their correctness is then stated in the abstract framework of $\lambda\sigma$-calculus [Cur89].

1 Introduction

Abstract Machines used to compile functional languages, stop when the weak head normal form of their argument is reached. Although, it is enough in most cases because the result belongs to a basic domain, it can be sometimes convenient to go further :

to optimize a program :

- by specializing a function to some of its arguments at *compile time* (partial evaluation),
- by specializing a function to some part of its input at *run time* (a parser generator to a grammar or a compiler generator to a denotationnal semantic of a language or a compiler to the text of a program) [App87].
- by unfolding recursive functions to coalesce them in a single one (folding / unfolding) [BD77].

to handle terms explicitely as a structure :

- when we use the Curry-Howard principle of equivalence between λ-terms and proofs in a program synthetiser. This work was originally done to execute programs extracted from the calculus of constructions where all the structures are coded with λ-terms [Moh86].
- when types are represented by λ-terms as in a second or higher order type-checker [ACCL90].
- for higher order logic programming [NW90].

First, we present an overview of the folklore of environment based machines: Krivine's machine is a simple example using call by need strategy. We will show the influence of De Bruijn's coding of variables and the link with $\lambda\sigma$-calculus.

Then, we will introduce a first version of a full reducer: KN. We will insist on the choice of the origin taken for our numbering system of variables. A second version that shares reductions between calls and lets the user control the evaluation is then proposed. It is in fact a kind of supervisor controlling different instances of KN.

We will give a sketch of the proof of the correctness of our machine in the abstract framework of the $\lambda\sigma$-calculus. It shows the power of this formalism to understand machines and the underlying assumptions made in machine's design. Finally, experimental results will be given with insights on how to implement those machines.

2 Background

2.1 De Bruijn's notation for variables in the λ-calculus

In classical λ-calculus, a variable of a closed λ-term consists of :

- a λ which declares it.
- a set of occurrences in the subterm of the declaration.

We usually use an identifier taken in an arbitrary set to visualize that link. But, we can also draw it with an arrow in a graphical representation.

De Bruijn extracted a textual form of this idea from the following remarks:

- there is a unique path from the root of a term to a given variable.
- the declaration is somewhere on that path and can be identified by the number of declarations between it and the occurrence. This number is called the De Bruijn's index of the occurrence.

For example:

$$\lambda x.x\,(\lambda y.x\;y) \longrightarrow \lambda 1\,(\lambda\, 2\, 1)$$

As the number of λ's between the root of a term, and an occurrence has a fixed value, it is in fact a pure matter of taste to choose the occurrence of the variable or the root as

*45 rue d'Ulm 75230 Paris CEDEX 5 France - email: cregut@dmi.ens.fr

the origin of numbering (We call the latter solution *reversed De Bruijn's indexing*). With reversed indexes, the previous term would give :

$$\lambda 1\ (\lambda\ 1\ 2)$$

The rule for β-reduction is simpler for local (bound in the argument of the redex) variables in the first case, for global (free) ones in the second one. Here is a description of β-reduction in De Bruijn's notation.

$$(\lambda.M)N \rightarrow \sigma_0(M,N)$$

$$\begin{aligned}
\sigma_n(M_1M_2,N) &= \sigma_n(M_1,N)\sigma_n(M_2,N)\\
\sigma_n(\lambda(M),N) &= \lambda.\sigma_{n+1}(M,N)\\
\sigma_n(m,N) &= \begin{cases} m-1 & \text{if } m>n\\ \tau_0^n(N) & \text{if } m=n\\ m & \text{otherwise}\end{cases}
\end{aligned}$$

where the translatory function τ_i^n is defined by :

$$\begin{aligned}
\tau_i^n(m) &= \begin{cases} m+n & \text{if } m\geq i\\ m & \text{otherwise}\end{cases}\\
\tau_i^n(M_1,M_2) &= \tau_i^n(M_1)\tau_i^n(M_2)\\
\tau_i^n(\lambda.M) &= \tau_{i+1}^n(M)
\end{aligned}$$

τ shifts indexes of global variables from the value of the De Bruijn's index of the substitued occurence (denoted by n).

We would use the same function σ with reversed indexing (it is not entirely by accident: there is a kind of symmetry). But there, global variables are denoted by small numbers. The rule would become:

$$C[(\lambda.M)N] \rightarrow C[\sigma_n(M,N)]$$

where n is the index of the λ(we depend on the context). In the second version, τ shifts the indexes of local variables.

2.2 Krivine's Abstract Machine

If we glance at a term from its root to an occurrence of a variable, collecting on a stack the value associated to each λ encountered (it can be anything if the abstraction is not part of a redex), then the value of the occurrence whose De Bruijn index is n is stored in the n^{th} cell of the stack. What has been built is called the environment of the term. The indexes are a way to code the access to the value of a variable via an environment.

But, β-reduction is tricky with De Bruijn's notation because it moves the indexes of global variables : something we don't want in compiled code. The idea is to freeze the environment of the term before reduction with the term itself in what we call a *closure*. So, when an internal redex is reduced, everything will be as if the external ones had not been touched.

Krivine's abstract machine follows that principle. It walks through the left spine of the term substituting variables only when they are accessed. A state of the machine is made of the representation of a term (a graph or a linearized form of it, we call nodes instructions because it is what they are actually when we have compiled the term), an environment which is a list of closures (we can't use arrays because we must share parts of the environments) and a stack to collect arguments before their associated λ is found.

We present a description of the K machine based on structural operational semantic [Plo81].

state ::= code * environment * stack

$$\begin{aligned}
((C_1\ C_2),e,p) &\rightarrow (C_1,e,< C_2 : e >:: p)\\
(\lambda(C),e,f :: p) &\rightarrow (C,f :: e,p)\\
(0,< C_2 : e_2 >:: e_1,p) &\rightarrow (C_2,e_2,p)\\
(n+1,f :: e,p) &\rightarrow (n,e,p)
\end{aligned}$$

For an input "C", the machine starts with $(C,[],[])$ and stops when no rule can be applied. It is close to the T.I.M. [FW87]. The main difference is that we don't require supercombinators, the trade-off is the representation of environments. Improvements designed for the TIM [Arg89] can be adapted to K. K can also be regarded as a lazy version of the CAM [CCMS86].

2.3 The $\lambda\sigma$-calculus

The $\lambda\sigma$-calculus is an extension of λ-calculus with explicit simultaneous substitutions (as opposed to individual substitutions of the previous formalism). The syntax of terms is De Bruijn's one with an added representation for closures. The $\lambda\sigma$-calculus has two assets over pure λ-calculus to prove the correctness of abstract machines :

- It provides a way to split substitution into basic components and so study its implementation.
- the way it hides the shift of indexes (done by τ in 2.1) in β-reduction is close to what is done in abstract machines.

The substitution part in a closure is really what we call the environment in a machine. The major difference between the theory and implementation is the environment of a term being directly deduced from the environment of its father node even when it is a substitution. This problem is solved in a machine by keeping a pointer on the related part of the environment.

$$\begin{aligned}
\text{Terms of } \lambda\sigma :\ A &::= p_i^n \mid (A^nB^n)^n \mid (\lambda A^{n+1})^n \mid (A^m \bullet \sigma_m^n)^n\\
\sigma_m^n &::= [A_1^n;\ldots;A_m^n]
\end{aligned}$$

There is a syntactical constraint on arities which is expressed by the upper indexes of the syntax. The arity expresses the number of free variables in a term. They will be understood in the following.
The rewriting rules of $\lambda\sigma$ are:

$$\begin{aligned}
\text{Beta} : (\lambda A)B^n &\rightarrow A \bullet [B;p_1^n;\ldots;p_n^n] &(1)\\
p_i^n \bullet [\sigma_1;\ldots;\sigma_n] &\rightarrow \sigma_i &(2)\\
(\lambda A)^n \bullet \sigma &\rightarrow\\
&\quad \lambda(A \bullet (p_1^{n+1} :: \sigma \bullet [p_2^{n+1};\ldots p_{n+1}^{n+1}])) &(3)\\
AB \bullet \sigma &\rightarrow (A \bullet \sigma)(B \bullet \sigma) &(4)\\
(A \bullet \sigma) \bullet \sigma' &\rightarrow A \bullet (\sigma \bullet \sigma') &(5)\\
A \bullet [p_1^n;\ldots p_n^n] &\rightarrow A &(6)
\end{aligned}$$

where $(\sigma \bullet \sigma')$ means $[\sigma_1 \bullet \sigma'; \ldots; \sigma_n \bullet \sigma']$ if $\sigma = [\sigma_1; \ldots; \sigma_n]$

Rules from 2 to 6 form a Noetherian system (called "Subst"). Translations from λ-terms to $\lambda\sigma$-terms and vice versa are obvious and will be written >>in both cases.

All the proofs on machines using $\lambda\sigma$ rely on the correctness of $\lambda\sigma$ with respect to β-reduction. It is stated by the following theorem :

Theorem 1 *Main theorem of $\lambda\sigma$-calculus (P.L. Curien)*

1. *If $M >> A$, then $A >> M (A \in \Lambda, M \in \Lambda\Sigma$*
2. *If $A >> M$ and $A \overset{Subst}{\rightarrow} A'$, then $A' >> M$*
3. *If $A >> M$, $A \overset{Beta}{\rightarrow} A'$ and $A' >> M'$, then $M \rightarrow M'$*
4. *If $M >> A, M \rightarrow M'$, and $M' >> A'$, then $A \overset{Beta, Subst^*}{\longrightarrow} A'$*
5. *If $A >> M >> A'$, then $A \overset{Subst^*}{\rightarrow} A'$.*

3 The KN machine

3.1 when does K stop ?

K machine without any added domain of constants can't give back any result but a closure. It stops when the stack is empty and the current instruction is a λ. This state represents the weak head normal form of the initial term (Outer redexes have been reduced until the term is either a variable or λM where M is a λ-term). It can't be a variable if the initial term is closed.

Now, we would like to get the full normal form of a term. We use the following rules to describe β-reduction on terms from elementary reduction of redexes (extracted from [Bar81]):

$$\begin{array}{rcll} (\lambda x.M)N & = & M[x := N] & (\beta) \\ M = N \; N = L & \Rightarrow & M = L & \\ M = N & \Rightarrow & MZ = NZ & \\ M = N & \Rightarrow & ZM = ZN & \textbf{(ri)} \\ M = N & \Rightarrow & \lambda x.M = \lambda x.N & (\xi) \end{array}$$

A WHNF-reducer carries out only the three first rules. But as **ri** is not used without ξ, the first problem is to cross the λ's.

3.2 carrying out rule ξ

First, as we are interested in the structure of terms, we will add an output to our machine viewed as a stream where we collect information on the normal form as it is built. The result will be a side effect of reduction.

Let us take a term in weak head normal form (WHNF).

To continue evaluation, we must fool the machine by giving it something to fill its environment. So we introduce a new node called "V". We record the presence of an unreduced abstraction by issuing a λ on the output.

If we continue evaluation and if the head normal form of the term exists, we will reach the call to the head variable of the term. It is an access to one of the dummy variables (or a constant if there are basic domains) stored in the environment. Two things must be performed:

- identify the variable.
- build the spine of the result and reduce the arguments of the head variable.

3.3 representing variables

There are two solutions to represent variables :

- We associate a unique name with each variable. It is issued with the λ, stored with the V node (as environment part) and given back with each occurrence of the variable.
- We use De Bruijn indexing. First, we define the *syntactic level of λ-nesting* which is the number of λ surrounding a subterm of the result. It is a kind of De Bruijn's numbering where the origin is the root of the term instead of a specific occurrence and it increases as we go down the tree.

  The index of the occurrence is the difference between the nesting of the occurrence and the level where the variable was introduced. This last number must be stored with the V-node.

3.4 rule ri

Dealing with arguments can take various forms. What is important to pursue evaluation is the closure itself and the curent level of nesting in the result. A first solution is giving back the head normal form to a meta-level controller that will decide whether to continue or not.

A second idea is producing immediately a syntactic result (a piece of text). This can easily be done with a simple system of marks on the stack. But we can also give back a tree if we record the address where the result of each evaluation must be stored. Suppressing recursion in such a function is a well known problem.

3.5 semantics

Here is the definition of data in use :

- Term = term of λ-calculus with De Bruijn indexes or "V(n)" where n is an integer,
- Environment = (Terme * Environment) List,
- Stack = (Term * Environment) List,
- State = Term * Environment * Stack * integer,

and the evaluation function :

- Ev: State $\rightarrow$ Term.

We use an exclamation mark to introduce results. So,

$$Ev(S) = N \Longleftrightarrow S \rightarrow !N$$

$$\begin{array}{rcl} ((C_1\, C_2), e, s, n) & \rightarrow & (C_1, e, < C_2 : e >:: s, n) \\ (\lambda(C), e, f :: s, n) & \rightarrow & (C, f :: e, s, n) \\ (m+1, (f :: e), s, n) & \rightarrow & (m, e, s, n) \\ (0, (< C' : e' >:: e, s, n) & \rightarrow & (C', e', s, n) \end{array}$$

$$\frac{(C, < V(n) : [\,] >:: e, [\,], n+1) \rightarrow !N}{(\lambda(C), e, [\,], n) \rightarrow !\lambda N} \; (\xi')$$

$$\frac{(c_i, e_i, [\,], n) \rightarrow !N_i (\forall i \in \{1 \ldots p\})}{(V(m), [\,], \begin{bmatrix} < c_1 : e_1 > \\ \vdots \\ < c_p : e_p > \end{bmatrix}, n) \rightarrow !(n-m)\; N_1 \ldots N_p} \; (\mathbf{ri}')$$

Notice that the first four rules are exactly those of th K-machine, while, as stressed by the names ξ', ri', the two additional rules carry out ξ and ri. If we only want to print the result, we can handle recursion explicitly with the following rule in place of the last two:

$$(\lambda(C), e, (>, k) :: s, n) \xrightarrow{\lambda} (C, < V(n) : [\,] >:: e, (>, k) :: s, n+1)$$

$$\begin{aligned}
(V(m), [\,], s, n) &\xrightarrow{(n-m)} (Sp, [\,], s, n) \\
(Sp, [\,], < c : e >:: s, n) &\xrightarrow{(} (c, e, (>, n) :: s, n) \\
(Sp, [\,], (>, k) :: s, n) &\xrightarrow{)} (Sp, [\,], s, k) \\
(Sp, [\,], [\,], n) &\rightarrow STOP
\end{aligned}$$

"Sp" is an instruction that handles the printing of the spine. $(>, n)$ is a mark put on the stack to record the level and to insert parenthesis where needed. The output is printed above the arrow. We initially print a "(" and start with $(C, [\,], [(>, 0)], 0)$ where C is the term to reduce.

4 The KNP machine

4.1 computing head normal forms

Now we would like to share common parts of the computation of variables among differents calls. First, we need a way to use results like source terms.

Then, we don't want to loose termination property for any term where it exists. So we can't completely reduce a variable without knowing what is needed. But, we can use the following fact [Bar75] [Lev78] :

Theorem 2 *the following statements are equivalent:*

- *M has a head normal form.*
- *M is solvable.*
- $\exists N_1 \ldots N_k$, *M* $N_1 \ldots N_k$ *has a normal form.*

Definition 1 *A context* $C[\,]$ *is* strict *if there exists A such that* $C[A]$ *has no normal form.*

A term F is solvable *if there exists a strict context* $C[\,]$ *such that* $C[F]$ *has a normal form.*

So, in the next machine, we won't use leftmost-redex strategy but normal parallel reduction with left call by value (see [BBKV76] or [Lev78]). (we can safely compute the head normal form of a function, update it and call it with its arguments).

4.2 closures as instructions

Now, we must keep a representation of the unevaluated part of the head normal form. An idea emphasized by $\lambda\sigma$-calculus is viewing closures as part of the syntax so as part of the instruction set. We can use them to represent variables in the environment but also unreduced part of the result. When such a closure will become the current code, we will follow exactly the same steps as for a variable. Our new machine KNP can be viewed as a controller calling recursively a KN machine on each closure. It is the surrounding context that controls how much of the function is required. Closures are like annotated tokens flowing down the term tree and transforming it.

4.3 handling free variables

We have to handle free variables in subterms. We have seen that De Bruijn's β-reduction handles them in another way and shifts them according to the number of λ that occur between the introduction and the occurrence of the variable. But, we have also seen that indexes of global variable are not changed with reversed indexing. This change of origin for numbering gives us a way to represent free variables independently from their use. In fact, it is also the numbering we have used for formal variables in the previous version.

But, a global variable of a subterm may be local in the surrounding term. We need a way to decide whether to change a global to a local or not. This choice depends on the value of i in our presentation of β-reduction in 2.1 (in the definition of τ). Here, it is the value of the current level when the argument is linked to its variable but it is also the current level when the variable was put on the stack. Now, we introduce a new index called the *global index* that records the nesting level of the whole variable. When a formal variable is found, it is translated in a local one or it is kept formal, depending on whether its index is greater than the global index or not.

If we had directly translated formal variables in local ones, we would have had disorders while updating as in this example.

$$\lambda t.(\lambda u.u(\lambda v.u))t$$

would lead to :

$$\lambda t.t(\lambda v.v)$$

because the index of t in the second occurrence of u would still be 1 which is wrong. Free variables are considered like constants and must not be translated in accesses to the environment.

Arguments of the head variable are wrapped into closures to keep the value of the global index, the value of the nesting level is usually the same as the value stored when the argument was put on the stack.

4.4 semantics

A state is a 5-tuple (code * environment * stack * nesting level * global level). The syntax of terms is the following:

$$E ::= \lambda(E) \mid E_1\ E_2 \mid i \mid V_i \mid < E : [E_1; \ldots; E_n], n, k >$$

the axioms and inference rules are :

$$((C_1\ C_2), e, p, n, k) \rightarrow (C_1, e, (< C_2 : e, n, n >:: p), n, k) \quad (7)(8)$$

$$(\lambda(C), e, (f :: p), n, k) \rightarrow (C, (f :: e), p, n, k) \quad (9)$$

$$(i, (f :: e), p, n, k) \rightarrow (i-1, e, p, n, k) \quad (10)$$

$$(0, (f :: e), p, n, k) \rightarrow (f, [\,], p, n, k) \quad (11)$$

$$\frac{(C, (V_n :: e), [\,], n+1, k) \rightarrow !N}{\lambda(C), e, [\,], n, k) \rightarrow !\lambda N} \quad (12)$$

$$\frac{(C', e', [\,], n', k') \rightarrow !N}{(< C' : e', n', k' >, e, p, n, k) \rightarrow (N, e, p, n, k)} \quad (13)$$

if $m \geq k$ then

$$(V_m, e, [p_1; \ldots; p_p], n, k) \rightarrow$$
$$!(n-m) \ < p_1 : [\,], n, k > \ldots < p_p : [\,], n, k > (14)$$

if $m < k$ then

$$(V_m, e, [p_1; \ldots; p_p], n, k) \rightarrow$$
$$!V_m \ < p_1 : [\,], n, k > \ldots < p_p : [\,], n, k > \quad (15)$$

The following rule is an easy improvement:

$$(C_1\ i), [e_1; \ldots; e_j], p, n, k) \rightarrow (C_1, [e_1; \ldots; e_j], (e_i :: p), n, k)$$

It prevents accumulating closures around a term when it is carried across recursive calls. The next rewriting rule may be tried after creating any closure:

$$<< C : e, n, k >: [\,], n', k + l > \rightarrow < C : e, n, k >$$

It just states that the term has only k free variables and further processing without environment can't change it.

4.5 how to get full normal form ?

A new mechanism is introduced to get the total reduction of the initial term. We can add strategies or heuristics at this level to decide whether we want to continue reduction or not. It can also be a classical K machine that shares the reduction of a specific function. Here is the simplest kind of controller :

- NF is a function that takes a partialy evaluated term and unwinds closures.
- KNP takes a term and gives back its normal form if it exists.

$$NF(i) \rightarrow i \quad (16)$$

$$\frac{NF(C) \rightarrow C'}{NF(\lambda(C)) \rightarrow \lambda(C')} \quad (17)$$

$$\frac{NF(A) \rightarrow A' \ \ NF(B) \rightarrow B'}{NF(A\,B) \rightarrow A'\,B'} \quad (18)$$

$$\frac{(C, e, [\,], n, k) \rightarrow !N \ \ NF(N) \rightarrow N'}{NF(< C : e, n, k >) \rightarrow N'} \quad (19)$$

$$\frac{NF(< C : [\,], 0, 0 >) \rightarrow N}{KNP(C) \rightarrow N} \quad (20)$$

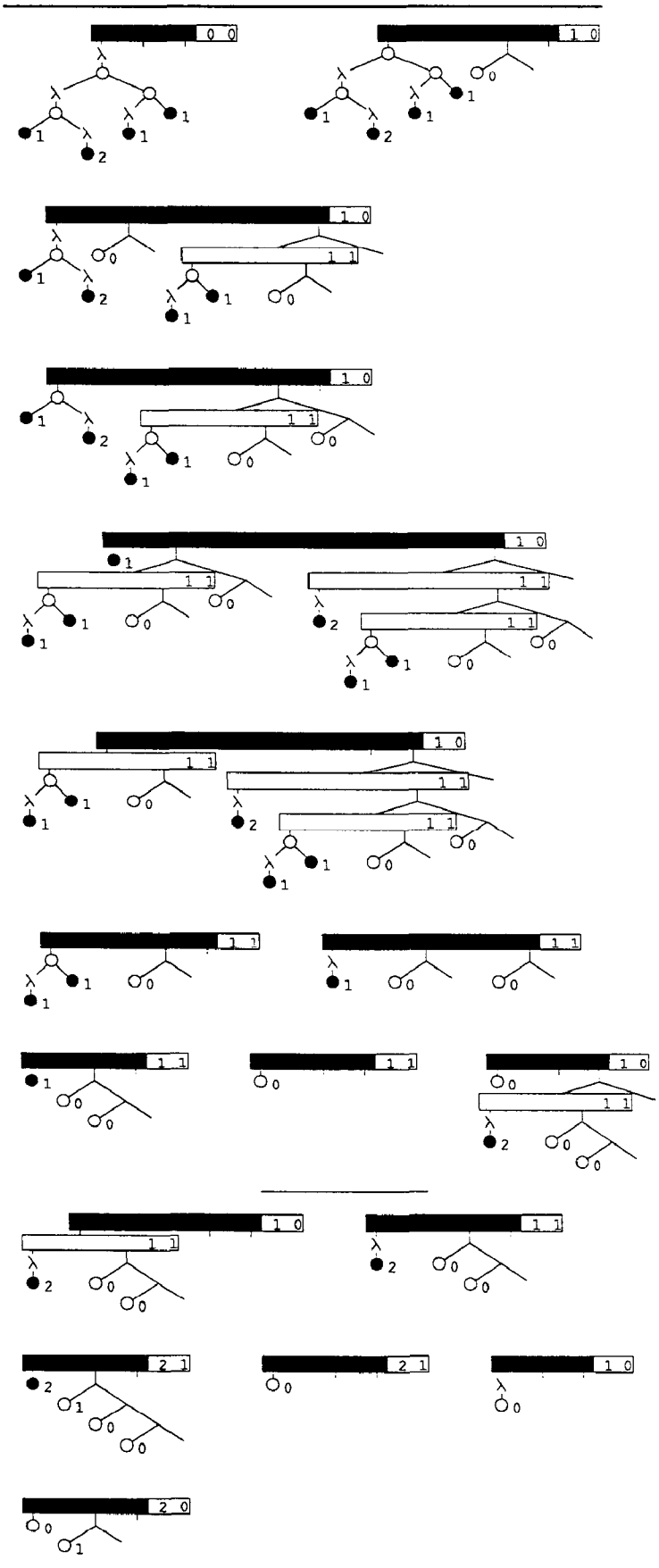
4.6 an example

We give a trace of the reduction of the following term :

$$\lambda t.(\lambda u.u(\lambda v.u))((\lambda x.x)t)$$

which gives back :

$$\lambda t.t(\lambda v.t)$$



The conventions used in the drawing are :

- applications are represented by a white circle,
- local variables by a black circle with the De Bruijn's index,
- global variables by a white circle with the reversed De Bruijn's index,
- closures by a white box with the term, its environment, the nesting level and the global index,
- the current state by a black box like a closure but the third term is the stack.

First we compute the head normal form of the term which is $\lambda t.t[]$. We can see how $((\lambda x.x)t)$ is updated in t in the last picture before the horizontal line. The second part is the reduction of the argument of the head variable t which gives back $(\lambda v.t)$.

5 Link between the $\lambda\sigma$-calculus and KNP

5.1 objects of the proof

There are two reasons to prove the corectness of those machines:

- to link theory and its application so that the implementation can be improved because it is better understood.
- to make sure that the rather complex handling of indexes is correct.

The $\lambda\sigma$-calculus provides a formal setting that factorizes common points of environment-based machines. So we can use it as a basis for our proofs and only deal with specific features of our abstract machine. See [HL86] for an example of proof of termination directly at the level of CAM combinators (a machine that is not very far from K).
First, we must understand the limit of $\lambda\sigma$-calculus for our goal :

- It is unable to express any kind of sharing,
- It has syntactical constraints that do not easily handle the V-nodes which are not easily embedded in $\lambda\sigma$ notation.
- If we don't want to use the classical orientation of the rules of the system "subst" in $\lambda\sigma$(that is necessary for KNP) we can't use that it is noetherian.

5.2 sketch of the proof of KNP

We just give the main ideas introduced to prove KNP execution is correct:

- First, we must adapt KNP so that the translation of V-nodes respects the arity rules. The main change is the use of identity substitutions instead of empty environments to keep arity of terms correct. The correctness of the proof in the $\lambda\sigma$-calculus will rely on the added variables but in fact, they are useless for transitions of machine states. Equations (11),(15) and (14) are respectively replaced with:

$$(0,(f :: e),p,n,k) \rightarrow (f,[V_{n-1};\ldots;V_0],p,n,k) \quad (21)$$

$$(V_m,e,[s_1;\ldots;s_p],n,k) \rightarrow \\ !(n-m) < s_1 : [V_{n-1};\ldots;V_0],n,k > \ldots \\ \ldots < s_p : [V_{n-1};\ldots;V_0],n,k > \quad (22)$$

$$(V_m,e,[s_1;\ldots;s_p],n,k) \rightarrow \\ !V_m < s_1 : [V_{n-1};\ldots;V_0],n,k > \ldots \\ \ldots < s_p : [V_{n-1};\ldots;V_0],n,k > \quad (23)$$

- Then, we introduce the translation function from states into $\lambda\sigma$-terms. The superscript n represents the level of free variables in the translation.

$$\begin{aligned} T^n(p_i) &= p_i \\ T^n(V_i) &= p_{n-i} \\ T^n(A\,B) &= T^n(A)\,T^n(B) \\ T^n(\lambda A) &= \lambda(T^{n+1}(A)) \end{aligned}$$

$$T^n(< C : e,m,k >) = T^{len(e)}(C) \bullet T^n(e) \bullet [p_1;\ldots \\ \ldots;p_{m-k};p_{n-k};\ldots;p_n]$$

$$T^n([e_1;\ldots;e_k]) = [T^n(e_1);\ldots;T^n(e_k)]$$

where

$$len([e_1;\ldots;e_n]) = n$$

The translation of a state at level n is :

$$T^n(C,e,[s_1;\ldots;s_l],m,k) = \\ (((T^{len(e)}(C) \bullet T^n(e))\,T^n(s_1)\ldots \\ \ldots\ T^n(s_l) \bullet Cor(m,n,k)$$

where $Cor(m,n,k)$ stands for :

$$[p_1;\ldots;p_{m-k};p_{n-k};\ldots;p_n]$$

Cor shifts the indexes as in the classical β-rule with De Bruijn's notation. Local variables are trapped by λ or closures. m is the level of the current occurence of the closure. k is the level that separates the locals from the globals. Cor shifts the global according to the difference between n which is the current level for closure processing and m.

- We prove the following theorem by induction on the number of reduction steps :

Theorem 3 *If $E =< C : e,n,k >$ is a well formed state.*

- *if $E \rightarrow E'$ then $T^m(E) \stackrel{\Lambda\Sigma^*}{\rightarrow} T^m(E')\ \forall m \geq k$*
- *if $E \rightarrow !N$ then $T^m(E) \stackrel{\Lambda\Sigma^*}{\rightarrow} T^m(N)\ \forall m \geq k$*

The key is the shape of the result obtained by evaluating a closure.

- Then, we go back to the initial machine and show that cells added for the proof in the environment are never accessed and don't change the access path to others.

So they are not necessary. We use the function "Access Depth" :

$$\begin{aligned} AD(p_i) &= i \\ AD(V_i) &= 0 \\ AD(< C : e, n, k >) &= n - k \\ AD(A\ B) &= \max(AD(A), AD(B)) \\ AD(\lambda(C)) &= \max(AD(C) - 1, 0) \end{aligned}$$

We use the shape of a closure result to justify the third case in the definition of AD.

- We study the termination of reduction :
  - The time between effective reduction is not infinite (we say that transitions that do not deal with λ (i.e. every rules but (9) and (12)) are administrative rules).
  - The machine follows a sure strategy in the choice of its redexes.

6 Practical issues

6.1 how to realize the machines

An operational rule with a numerator and a denominator can be viewed as a subroutine call. The result is built from the result of the subroutine call and the current state.

It is a side effect of the reduction process and can be directly issued on the output stream if we just want to display it. We may need use the stack to keep intermediate structures if we want further processing (in the second machine we want to get some new code).

We can notice that only one stack is used at a time so we can share a single real stack by using sentries to mark ends of local stacks. There are few pointers that describe a state, so recursion can be easily suppressed.

In the second version, using compiled code is probably not the right solution :

- structure of terms evolves too much during evaluation with closure updating: it requires indirection nodes and physical updating in the middle of the code would be required.
- we would like to make garbage collection of dead code. The amount of produced code is not fixed as it is in a classical ML implementation and most of it becomes useless when the function is reduced.

Nodes may be tagged with the address of their corresponding code to increase the evaluation speed (see [PJ87]). We also need a level of indirection on closures to update them (it seems hard to update directly a closure node if both codes (the old and the new ones) don't require the same space).

6.2 preliminary results on time requirements

Here is a good example of the power of KNP : If $\overline{n}$ denotes the n^{th} Church numeral then the computation of $\overline{2}\overline{n}I$ is linear with KNP and exponential with KN. Other machines (CAM,TIM,FAM,G-machine...) would be as slow as KN (To get a result imagine the term is applied to an identity which makes a side-effect on the output) because they update their arguments with their weak-head normal form and not their truly reduced value.

But KNP may also be slower when recursive functions computing growing results are used because it can't be seen that the result is just copied when the machine carries it across the different levels of function calls.

A good example is this function computing the predecessor for Church numerals which becomes quadratic instead of linear.

$$\lambda nfx.((n(\lambda e(cons(e\ snd)(f(e\ snd))))(cons\ x\ x))fst)$$

where $fst = \lambda xy.x$, $snd = \lambda xy.y$ and $cons = \lambda xyf.f\ x\ y$. Optimizations given at the beginning of the article in 4.4 may reduce that cost by preventing the building of useless closures but fails to detect all the useless computation. Nevertheless, we've got a limit on the induced slowing-down.

Theorem 4 *Let (t_n) be a sequence of λ-terms. If the complexity for reducing t_n is an $O(f(n))$ with KN where $f(n) \to \infty$ for $n \to \infty$, then complexity of KNP on this sequence can't be more than an $0(f^2(n))$.*

It results directly from the following facts :

- there can't be more occurences of variables in use than reduction steps
- an occurence has three stages in its life :
  - it is free in the computed subterm (requires constant time).
  - it is then bound and reduced (once and once only for the variable). The time required is the same as for KN because we only reduce what is needed.
  - it is then used for reduction at upper level but this part is numbered in the processing of the variables corresponding to those level.

7 Conclusion

7.1 improvements

First, we can introduce constants in the same way as global variables. If they have computational effects (if we want to apply operators to those constants), we can deal with them with continuation passing style as in TIM. The real problems come from the rules used to reduce partially formal expressions. As the vision of terms is rather local in a practical abstract machine, it may be difficult to realize global alteration of the code. These rules can also be handled in a completely different way through an interface with a symbolic reducer.

Recursion can be efficiently handled with an explicit construct ("let rec" of ML) in exactly the same way as for CAM by making environments of recursive terms point directly on themselves. Looping environments don't create interferences with numbering systems because the result is still without loops.

7.2 new directions

One of the most interesting point of KN is its simplicity. We can easily mix it with K which is an efficient machine especially when we use the classical techniques of sharing developed for TIM (see [FW87]). All we have to change is the handling of the empty stack.

KNP however is easy to control and when mixed with K let us choose which part of the term must be developped. It

can be the basis of a partial evaluator. One of our goal is to be abble to control reduction until we find a point where folding is interesting [Wad88]. As we can't rely on termination of term enumeration, help of the user and heuristics will be required to control the process.

It could be interesting to develop a language where the syntax of terms and of the control of their reduction would be expressed in a single formalism so that terms and control over them could be compiled in the same way. A system of tokens flowing down the terms controlled by marks could be an answer.

I thank Guy Cousineau for following my work, Pierre-Louis Curien for having introduced me to $\lambda\sigma$-calculus and pointing out relevant questions on indexes handling and Emmanuel Chailloux for his advices on the use of CAML to do drawings.

References

[ACCL90] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitution. In *POPL*, 1990.

[App87] Andrew W. Appel. *Reopening Closures.* Technical Report TR 079-87, Princeton university, 1987.

[Arg89] Guy Argo. Improving the three instruction machine. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, ACM, Addison-Wesley, London, 1989.

[Bar75] H.P. Barendregt. *The type free λ-calculus.* Technical Report, Utrecht University and ETH Zurich, 1975.

[Bar81] H. Barendregt. *The Lambda Calculus.* North Holland, 1981.

[BBKV76] H.P. Barendregt, J. Bergstra, J.W. Klop, and H. Volken. *Degrees, reductions and representability in the λ-calculus.* Technical Report 22, University of Utrecht, 1976.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1), 1977.

[CCMS86] Guy Cousineau, Pierre-Louis Curien, Michel Mauny, and Ascander Suárez. *Combinateurs Catégoriques et Implémentation des Langages fonctionnels.* Technical Report 86-3, LIENS, 1986.

[Cur89] Pierre-Louis Curien. The strong calculus of closures or $\lambda\sigma$-calculus a short note. 1989.

[FW87] John Fairbairn and Stuart Wray. Tim : A simple, lazy, abstract machine to execute supercombinators. In *LNCS 274*, Springer Verlag, 1987.

[HL86] Thérèse Hardin and Alain Laville. Proof of termination of the rewriting system subst on c.c.l. *Theorical Computer Science*, 46, 1986.

[Kri85] Jean-Louis Krivine. Un interpréteur du λ-calcul. 1985.

[Lev78] Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda-calcul.* PhD thesis, Université Paris VII, 1978.

[Moh86] Christine Mohring. *Exemples de dévelopement de Programmes dans la Théorie des Constructions.* Technical Report 497, INRIA, 1986.

[NW90] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *this conference*, 1990.

[PJ87] Simon Peyton-Jones. *Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Plo81] Gordon D. Plotkin. *A Structural Approach to Operational semantics.* Technical Report DAIMI FN-19, Aarhus University, 1981.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP*, 1988.