Theoretical Computer Science 82 (1991) 389-402
Elsevier

Note

An abstract framework for environment machines

P.-L. Curien

LIENS-CNRS, Ecole Normale Supérieure, 45 Rue d'Ulm, 75230 Paris Cedex 05, France

Communicated by M. Nivat
Received July 1990

Abstract

Curien, P.-L., An abstract framework for environment machines (*Note*), Theoretical Computer Science 82 (1991) 389-402.

We present a nondeterministic calculus of closures for the evaluation of λ-calculus, which is based, not on symbolic evaluation (β-reduction), but on the paradigm of environments and closures, as in the old SECD machine, or in the more recent Categorical Abstract Machine (CAM). This calculus stands as a suitable abstraction right above those devices: there is no commitment as to the order of evaluation, and no explicit handling of stacks: the calculus is expressed in the style of Structural Operational Semantics, by a set of conditional rewrite rules. The CAM, and a very simple lazy abstract machine, due to J.-L. Krivine, arise naturally by first restricting the calculus to a specific strategy, and then implementing recursive calls by a stack. The Church-Rosser property and termination in presence of types hold in the calculus of closures, and are easier to establish than in the λ-calculus. The calculus of closures has served as a starting point to a more powerful general calculus of substitutions at which we hint shortly, insisting on its category-theoretic significance.

1. Introduction

The basic feature of most current realistic implementations of λ-calculus-based functional programming languages is the reduction to closed weak head-normal forms (i.e. abstractions or constants of basic type).

Another basic feature of all implementations (even of those which attempt to be based properly on the full β-reduction) is that substitution is not a magic, metalevel,

one-step operation, but a process which has to be carried along the structure of the substituted term. Even if the data structure shares the occurrences of a bound variable, copying or "peeling off" of (some of) a body of a function is commonly performed before the application to an argument in order to preserve sharing with other arguments. There should be a trace of this cost in the syntax.

Those implementations which are based on environment machines also share another feature: no substitutions are carried under λ's. In those implementations, the environment is kept separate from the evaluated program, so that the value of a function is naturally a closure, which has a code part, and an environment part. In this paper we define and investigate a calculus of closures.

We shall show that it enjoys the Church–Rosser property, and we shall demonstrate how it can serve as a useful intermediate between λ-calculus and actual abstract machines. We shall define a simply typed calculus of closures and show that it strongly terminates.

The main virtues of this calculus are its simplicity and its heuristic character: the structure of abstract machines can be guessed almost deterministically from the abstract specification of strategies in the calculus of closures.

Our calculus has close relations with the weak λ-calculus. What is most commonly understood under this name is the formal system obtained by removing the ξ-rule from the formal system of λ-calculus (that is, the abstraction congruence rule). We are thus left with the following system:

$$(\lambda x.M)N \to M[N/x],$$

$$\frac{M \to M'}{MN \to M'N},$$

$$\frac{N \to N'}{MN \to MN'}.$$

The weak λ-calculus has a severe drawback: unlike the calculus of closures presented here, it is not confluent. Indeed, consider the term $(\lambda yx.y)(II)$ (where I is $\lambda x.x$). We can either reduce the root and obtain $\lambda x.II$ or reduce II and obtain $(\lambda yx.y)I$, which further reduces to $\lambda x.I$. But, since ξ has been removed, there is no reduction from $\lambda x.II$ to $\lambda x.I$. For this reason, only specialized, deterministic weak calculi of β-reduction have been investigated so far. For instance, the lazy λ-calculus [2, 21] restricts to the deterministic lazy strategy:

$$\frac{M \overset{*}{\to}_l \lambda x.P}{MN \to_l P[x \leftarrow N]}.$$

A general calculus of explicit substitutions, simulating unrestricted β-reductions, has been developed before the present work, in the framework of categorical combinators [7], and after this work, in the framework of $\lambda\sigma$-calculus [1, 8]. We

shortly discuss here a variant of the $\lambda\sigma$-calculus which has close relations with our calculus of closures, and has a direct category-theoretic significance. However the versions presented in [1], and especially in [8] are more suited for a thorough syntactic investigation, both of strong and weak calculi of explicit subsitutions.

In Section 2 we present the calculus of closures and show that it is confluent. In Section 3 we derive simple abstract machines from lazy and eager computation strategies. In Section 4, we introduce a simply typed calculus of closures, and prove that it is strongly normalizing, by a very simple version of the computability method. In Section 5 we briefly hint at a general calculus of substitutions, and propose a variant which has a nice category theoretical foundation.

2. The calculus of closures

The distinctive feature of our calculus is that it gives a first-class status to closures. The syntax is as follows. There are two sorts: terms and closures. The terms are those of De Bruijn's λ-calculus:

$$\textit{Terms} \quad M ::= n \mid M\,M \mid \lambda M,$$

$$\textit{Closures} \quad u ::= M[\rho]$$

where ρ stands for a finite list $u_1; \cdots; u_n$ of closures. For the readers familiar with [1], the correspondence with $\lambda\sigma$-notation is as follows: the empty list corresponds to id, and $u_1; \cdots; u_{n+1}$ corresponds to $u_1 \cdot \ldots \cdot u_{n+1} \cdot id$. We shall thus employ the notation $u_1 \cdot [u_2; \cdots; u_{n+1}]$ for $u_1; u_2; \cdots; u_{n+1}$. We shall also write freely either $u_1; \cdots; u_n$ or $[u_1; \cdots; u_n]$. Priorities are as follows: $M\,N[\rho]$, $\lambda M[\rho]$ stand for $(M\,N)[\rho]$, $(\lambda M)[\rho]$. Computations are performed on closures only. They are specified by the following conditional system, which we call $\lambda\rho$.

Eval $$\frac{M[\rho] \xrightarrow{*} \lambda P[\nu]}{(M\,N)[\rho] \to P[N[\rho] \cdot \nu]},$$

Access $$n[u_1; \cdots; u_m] \to u_n \quad (n \leq m),$$

Env $$\frac{u_1 \xrightarrow{*} v_1 \cdots u_n \xrightarrow{*} v_n}{M[u_1; \cdots; u_n] \to M[v_1; \cdots; v_n]}.$$

There is no rule for abstraction, and the last rule, which expresses the congruence, introduces nondeterminism. The rest of the section is devoted to showing the Church–Rosser property for this system, and to exhibit some kind of standardization theorem for it.

Our inductions will work on a simple measure of complexity of derivations, which is defined as follows:

- multiple step derivation: sum of the complexities of each step
- (Eval): complexity of antecedent +1

- (Access): 1
- (Env): max of complexities of antecedents.

We write $\mathcal{K}(u \stackrel{*}{\to} v)$ for the complexity of $u \stackrel{*}{\to} v$. Strictly speaking, the complexity is associated, not with $u \stackrel{*}{\to} v$, but with the proof of the judgement $u \stackrel{*}{\to} v$. We shall not do it formally, but the reader should keep this in mind while reading the proofs. One easily checks that if $u \stackrel{*}{\to} v$ has complexity 0, then $u \equiv v$.

Theorem 2.1. *If $u \stackrel{*}{\to} v_1$ and $u \stackrel{*}{\to} v_2$, then for some v one has $v_1 \stackrel{*}{\to} v$ and $v_2 \stackrel{*}{\to} v$. Moreover $\mathcal{K}(v_1 \stackrel{*}{\to} v) \leq \mathcal{K}(u \stackrel{*}{\to} v_2)$, and $\mathcal{K}(v_2 \stackrel{*}{\to} v) \leq \mathcal{K}(u \stackrel{*}{\to} v_1)$.*

Proof. The proof is by induction on the lexicographic ordering $(\mathcal{K}(u \stackrel{*}{\to} v_1) + \mathcal{K}(u \stackrel{*}{\to} v_2), |u|)$, where $|u|$ is the size of u. We first remove from both derivations the steps of complexity 0. The resulting derivations have still the same length and the same endpoints. Suppose now that at least one of these derivations has more than one step, say $u \to v \stackrel{*}{\to} v_1$. Then we can apply induction on $u \to v$ and $u \stackrel{*}{\to} v_2$: we get, that, for some v', $v \stackrel{*}{\to} v'$ and $v_2 \stackrel{*}{\to} v'$, and that $\mathcal{K}(v \stackrel{*}{\to} v') \leq \mathcal{K}(u \stackrel{*}{\to} v_2)$. We can thus apply also the induction hypothesis to $v \stackrel{*}{\to} v_1$ and $v \stackrel{*}{\to} v'$. We are now reduced to the case where both derivations are one-step. The conclusion is obvious when u rewrites in both directions identicallly. If both reductions use (Env), then we use the second component of the lexicographic ordering, which decreases while the first component does not increase. If one reduction (say $u \to v_1$) is by (Env) and the other by (Access), the confluence diagram is easily closed with $v_1 \to v$ by (Access) and $v_2 \to v$ by (Env).

There are only two cases left: (Env)-(Eval) and (Eval)-(Eval). We examine (Env)-(Eval) first. We have $u \equiv M\,N[\rho]$, $v_1 \equiv M\,N[\rho']$ and $v_2 \equiv P[N[\rho] \cdot \nu]$. We can apply induction to $M[\rho] \to M[\rho']$ and $M[\rho] \stackrel{*}{\to} \lambda P[\nu]$. Thus for some ν' we have $\lambda P[\nu] \stackrel{*}{\to} \lambda P[\nu']$ and $M[\rho'] \stackrel{*}{\to} \lambda P[\nu']$. We then observe that the derivation $\lambda P[\nu] \stackrel{*}{\to} \lambda P[\nu']$ can only be composed of (Env) steps. We can turn it into a single-step (Env) without increasing the complexity. Thus its antecedents have complexities which are not greater than the max of the complexities of the antecedents of $M[\rho] \to M[\rho']$, i.e. of $N[\rho] \to N[\rho']$. From this we infer that the one-step (Env) derivation $P[N[\rho] \cdot \nu] \to P[N[\rho'] \cdot \nu']$ does not have a greater complexity than $M\,N[\rho] \to M\,N[\rho']$. It is routine to check the rest of this case.

The final case is (Eval)-(Eval). We have then $u \equiv M\,N[\rho]$, $v_1 \equiv P[N[\rho] \cdot \nu]$ and $v_2 \equiv P[N[\rho] \cdot \nu']$. We can use induction on the antecedents $M[\rho] \stackrel{*}{\to} \lambda P[\nu]$ and $M[\rho] \stackrel{*}{\to} \lambda P[\nu']$. Thus for some u' we have $\lambda P[\nu] \stackrel{*}{\to} u'$ and $\lambda P[\nu'] \stackrel{*}{\to} u'$. This forces $u' \equiv \lambda P[\nu'']$ for some ν'' and as already argued above, we can safely assume that these two derivations to u' are one-step and use (Env). One then has $P[N[\rho] \cdot \nu] \to P[N[\rho] \cdot \nu'']$ and $P[N[\rho] \cdot \nu'] \to P[N[\rho] \cdot \nu'']$, which settles the case (the complexity conditions are easily checked). □

We next establish a kind of standardization result. We show that in some sense, made precise in the next statement, lazy reduction always does better than any

reduction. The lazy restriction $\to_l$ of $\lambda\rho$ is defined by removing the rule (Env) from $\lambda\rho$:

LEval $$\frac{M[\rho] \stackrel{*}{\to}_l \lambda P[\nu]}{MN[\rho] \to_l P[N[\rho] \cdot \nu]},$$

LAccess $n[u_1; \cdots; u_m] \to_l u_n \quad (n \leq m)$.

Lemma 2.2. *If* $u \stackrel{*}{\to} \lambda P[\nu]$ *then* $u \stackrel{*}{\to}_l \lambda P[\nu']$ *and* $\lambda P[\nu'] \stackrel{*}{\to} \lambda P[\nu]$ *for some* ν', *and moreover* $\mathcal{K}(\lambda P[\nu'] \stackrel{*}{\to} \lambda P[\nu]) \leq \mathcal{K}(u \stackrel{*}{\to} \lambda P[\nu])$.

Proof. First notice that the derivation $\lambda P[\nu'] \stackrel{*}{\to} \lambda P[\nu]$ can only be composed of (Env) steps. The proof is by induction on the complexity of $u \stackrel{*}{\to} \lambda P[\nu]$. If this complexity is 0, one must have $u \equiv \lambda P[\nu]$, and the statement is obvious with $\nu \equiv \nu'$. We may thus assume that there is at least one step.

We look at the first step $u \to u'$. If it is (Access), it is a lazy step, and can be prefixed to the lazy derivation from u', which exists by induction. If it is (Eval), i.e. $u \equiv MN[\rho]$ and $u' \equiv Q[N[\rho] \cdot \nu_1]$, with antecedent $M[\rho] \stackrel{*}{\to} \lambda Q[\nu_1]$ of complexity $\mathcal{K}$, then we can apply induction to the antecedent: for some ν_1' one has $M[\rho] \stackrel{*}{\to}_l \lambda Q[\nu_1']$ and $\lambda Q[\nu_1'] \stackrel{*}{\to} \lambda Q[\nu_1]$, and $\mathcal{K}(\lambda Q[\nu_1'] \stackrel{*}{\to} \lambda Q[\nu_1]) \leq \mathcal{K}$. Thus we can decompose $u \to u'$ into $MN[\rho] \to_l Q[(N[\rho] \cdot \nu_1')] \stackrel{*}{\to} Q[N[\rho] \cdot \nu_1]$, where

$$\mathcal{K}(Q[N[\rho] \cdot \nu_1'] \stackrel{*}{\to} Q[N[\rho] \cdot \nu_1]) = \mathcal{K}(\lambda Q[\nu_1'] \stackrel{*}{\to} \lambda Q[\nu_1]) \leq \mathcal{K}.$$

We conclude this case by applying induction to the derivation from $Q[N[\rho] \cdot \nu_1']$ to $\lambda P[\nu]$, via $Q[N[\rho] \cdot \nu_1]$, remarking that it has a smaller complexity than $MN[\rho] \stackrel{*}{\to} \lambda P[\nu]$, since $\mathcal{K}(MN[\rho] \to Q[N[\rho] \cdot \nu_1]) = \mathcal{K} + 1$.

Let us suppose finally that $u \to u'$ is an (Env) step. If the whole derivation $u \stackrel{*}{\to} \lambda P[\nu]$ consists of (Env) steps only, then M must have the form $\lambda P[\nu']$, and the result is obvious. Otherwise, let $u'' \to u'''$ be the first non (Env) step of the derivation. Notice that we may transform $u \stackrel{*}{\to} u''$ into a single step reduction $u \to u''$ (still by (Env)), which does not have a greater complexity. Consider now the next step $u'' \to u'''$. If it is an (Access) step, we notice that we can apply (Access) first obtaining, say u_1'', and from there reach u''' by one of the antecedents of the (Env) rule; moreover, $\mathcal{K}(u \to u_1'' \to u''') \leq \mathcal{K}(u \to u'' \to u''')$. So we can go back to the case where the first step is (Access), which has been treated earlier in the proof. Similarly, if $u'' \to u'''$ is an (Eval) step, we can factor the two steps $u \to u''$ and $u'' \to u'''$ into a single (Eval) step. Let $u \equiv MN[\rho]$, $u'' \equiv MN[\rho']$, and $u''' \equiv Q[N[\rho'] \cdot \nu]$. We have $M[\rho] \to M[\rho']$ with the same complexity, say $\mathcal{K}_1$, as $MN[\rho] \to MN[\rho']$. If $\mathcal{K}_2$ is the complexity of the antecedent $M[\rho'] \stackrel{*}{\to} \lambda Q[\nu]$ of $u'' \to u'''$, then one easily checks that both derivations $u \to u'' \to u'''$ and $u \to u'''$ have the same complexity $\mathcal{K}_1 + \mathcal{K}_2 + 1$. So we can go back to the treatment of the case where the first step is (Eval). This ends the proof. □

We refer to [8] for a slightly different weak calculus, which presents the advantage of almost falling in the category of rewriting systems without critical pairs, and thus enjoys many nice properties "for free".

In the next section, we shall show how abstract machines can be derived from the lazy and eager strategies in the calculus of closures.

3. Towards abstract machines

The implementations of λ-calculus roughly fall in two classes, according to the way of handling substitution. In graph reduction machines, substitution is performed by circulating terms to the appropriate leaves. The environment is built in the graph: this is the common paradigm of the graph reduction machines for the λ-calculus, Curry's combinators or supercombinators. The *G*-machine, developed at Göteborg, is a sophisticated example of this class of machines. In environment machines, on which we focus here, the environment is kept separate from the program to be evaluated, and is accessed as a kind of database.

Another apparent difference between these two classes of devices is that combinator reduction machines "carry substitutions under λ's", while the environment machines do not. For instance, the translation of $(\lambda xy.M)N$ is reduced to the translation of $\lambda y.(M[x \leftarrow N])$ in a combinator machine, while an environment machine keeps as a closure the code $\lambda y.M$ on one side and the environment where "x is N" on the other side. But this difference has nothing to do with the computing device, but rather with the way of compiling. The inputs of graph reduction machines are combinator expressions, and general λ-expressions have to be compiled by λ-lifting, and this phase is actually responsible for the difference just quoted. The inputs for environment machines can be general λ-expressions, but the compilation into combinators can be prefixed to environment machine execution (as in the TIM machine [9]), and in this case substitutions are carried under λ's as well.

We shall first demonstrate how the lazy and eager strategies in the calculus of closures lead naturally to two simple abstract machines. The lazy and eager machine we arrive at are respectively Krivine's abstract machine [11], and (a variant of) the Categorical Abstract Machine. Those machines differ from the strategies by the use of a stack, which implements recursive calls specified by the strategy (those calls are implicit in the hypotheses of the form $u \stackrel{*}{\rightarrow} v$).

We begin by "deriving" Krivine's machine (hereafter *K*-machine) from the lazy strategy $\rightarrow_l$. Our account is reasonably detailed, by lack of a published reference. Let us recall here the specification of the lazy strategy in the calculus of closures:

LEval $$\frac{M[\rho] \stackrel{*}{\rightarrow}_l \lambda P[\nu]}{M\,N[\rho] \rightarrow_l P[N[\rho] \cdot \nu]}\ ,$$

LAccess $n[u_1; \cdots; u_m] \rightarrow_l u_n (n \leq m)$.

A mechanical procedure to implement this strategy can be described as follows. Two pointers A, B move in two graphs, one for the term M, the other for the environment ρ: the first graph remains fixed, while the second keeps growing (the

garbage collector is, unfortunately, not part of the theory, so far). The interpreter keeps following the left spine of M, accumulating the arguments (which are built into closures with the current environment) in the stack, until an abstraction or a variable is met. If it is an abstraction, and if the stack is not empty, its body is evaluated in the enlarged environment obtained by consing the environment with the closure on the top of the stack. If a variable is reached, and if the environment has enough components, then its value, a closure, is fetched, and its evaluation is resumed, unless it is a constant, which is the result of the computation. In Fig. 1 a table is shown representing the K-machine (we use $u :: S$ to denote the result of pushing u on the top of S):

K-machine

Environment	Term	Stack	Environment	Term	Stack
ρ	MN	S	ρ	M	$N[\rho] :: S$
ρ	λM	$u :: S$	$u \cdot \rho$	M	S
$u \cdot \rho$	$n+1$	S	ρ	n	S
$M[\nu] \cdot \rho$	1	S	ν	M	S

Fig. 1.

Notice that normal form states of this machine are either of the form $(\rho, \lambda M, [\,])$ or of the form $([\,], n, S)$. They correspond to the two possible forms of weak head-normal forms in the λ-calculus: $\lambda x.M$ and $xM_1 \ldots M_n$. Much efficiency is gained by abandoning the view of a pointer moving inside the term (which supposes that the graph of the term has been built): instead the term can be viewed as code.

One should be able to formulate and to prove the correctness of this machine w.r.t the $\rightarrow_l$ strategy of the calculus of closures along the same lines as in [6], where the same program was carried for the CAM and the $\rightarrow_v$ strategy.

In presence of constant closures a, one needs to add the following rule to the abstract machine:

$a \cdot \rho$	1	[]	a	[]

The emptiness of the code part expresses that the resulting state is a normal-form state, and that the result of the computation is a.

When the attention s restricted to the reduction of supercombinator expressions, the second line in the description of the K-machine can be modified as follows:

ρ	$\lambda^n(M)$	$u_1 :: \cdots :: u_n :: S$	$u_1; \cdots; u_n$	M	S

The point is that $\lambda^n(M)$ is closed, thus does not need ρ. An interesting consequence is that the size of environment (n) is predictable from the code, thus one may easily allocate a vector $[u_1; \cdots; u_n]$, which allows for efficient variable access. With this optimization, the K-machine, executed on supercombinators, is very close to the Three Instructions Machine [9].

We now derive an abstract machine for eager evaluation, following the same line. First we specify an eager strategy for the calculus of closures:

$$\textbf{EEval}\quad \frac{M[\rho]\to_e \lambda P[\nu] \quad N[\rho]\to_e u \quad P[u\cdot\nu]\to_e v}{MN[\rho]\to_e v},$$

EAccess $n[u_1;\cdots;u_m]\to_e u_n \quad (n\leqslant m)$.

Notice that in this strategy there is only a "one-step" reduction: the reduction to the eager normal form. The implementation of (EEval) is slightly more complicated than the implementation of (LEval). One has to implement two recursive calls, one for the function M, and one for the argument N. This necessitates the use of two different kinds of markers in the stack. The markers will be called "L" and "R". The stack now receives closures and markers.

An eager machine

Environment	Term	Stack	Environment	Term	Stack
ρ	MN	S	ρ	M	$L :: N[\rho] :: S$
ρ	λM	S			$\lambda M[\rho] :: S$
$u\cdot\rho$	$n+1$	S	ρ	n	S
$u\cdot\rho$	1	S			$u :: S$
		$u :: L :: N[\rho] :: S$	ρ	N	$R :: u :: S$
		$u :: R :: \lambda M[\rho] :: S$	$u\cdot\rho$	M	S

Fig. 2.

The machine in Fig. 2 is essentially the Categorical Abstract Machine (CAM), the description of which is recalled in Fig. 3 (it is convenient to use the concrete syntax $(M \mathbin{@} N)$ for MN).

Notice that the CAM environment is not always an environment in the sense of the $\lambda\sigma$-calculus: it is not always a list of closures (see for example the right-hand side of the last rule). The stack of the CAM does not accumulate closures, but CAM "environments", i.e. terms and substitutions. In the third rule, the code concatenation MC can be avoided by pushing the code C on the stack. Thus, code-saving in the CAM takes place after the evaluation of both parts of an application, while in the

Categorical abstract machine

Environment	Code	Stack	Environment	Code	Stack
u	$(C$	S	u	C	$u :: S$
u	$@C$	$v :: S$	v	C	$u :: S$
v	$)C$	$\lambda M[u] :: S$	(u, v)	MC	S
u	$\lambda(M)C$	S	$\lambda M[u]$	C	S
(u, v)	$(n+1)C$	S	u	nC	S
(u, v)	$1C$	S	v	C	S

Fig. 3.

eager machine presented in Fig. 2, code is saved when entering an application node. The role of the markers L and R are played respectively by "@" and ")".

In [6] lazy evaluation was introduced "on the top" of call-by-value evaluation, with an explicit use of special instructions "*freeze*" and "*unfreeze*", which the compiler has to introduce where appropriate to force the evaluation to be lazy. This style had been advocated by Plotkin in [22]. The K-machine provides a more direct implementation of lazy evaluation, but it is not essentially different from the lazy CAM. We briefly show where these machines differ. The lazy compilation of application, as proposed in, say [6] (see [18] for more details) is

$$\mathit{Compile}(M\,N) = \text{"("}\ \mathit{Compile}(M)\ \text{"@"}\ \mathit{freeze}(\mathit{Compile}(N))\ \text{")"}.$$

Now a simple analysis shows some redundancy. Suppose the environment is u when the machine is ready to execute this code. First "(" saves u; later, "@" reintroduces the same u, and is immediately followed by the "*freeze*" instruction, which constructs the closure $N[u]$. We can avoid "(" and "@" by anticipating the freezing, and save N together with u upon entering the application node. This modified version of the lazy CAM is then essentially the K-machine.

4. Strong normalization for typed closures

In this section we introduce simply typed closures, and prove the strong normalization property for this calculus. We apply Tait's computability method in a particularly simple setting. The structure of simple types is defined as follows: there are basic types, with generic name κ, and if σ, τ are types, then $\sigma \Rightarrow \tau$ is a type. In a simply typed λ-calculus, there are no pure closed terms of basic type. To remedy this, we shall augment the $\lambda\rho$ calculus with a nonempty set of constant closures, of generic name a, for each basic type κ. These constants have thus a predefined type. Notice that constants appear at the level of closures, not of terms. We also limit the calculus to closed closures.

Here are the typing rules:

$$\sigma_1, \ldots, \sigma_n \vdash i : \sigma_i \quad (i \leqslant n),$$

$$\frac{\Gamma \vdash M : \sigma \Rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau},$$

$$\frac{\sigma, \Gamma \vdash M : \tau}{\Gamma \vdash \lambda M : \sigma \Rightarrow \tau},$$

$$\vdash a : \kappa \quad \text{predefined}$$

$$\frac{\sigma_1, \ldots, \sigma_n \vdash M : \tau \qquad \vdash u_1 : \sigma_1 \cdots \vdash u_n : \sigma_n}{\vdash M[u_1; \cdots; u_n] : \tau}.$$

Notice that in the name-free syntax a typing judgement for terms is of the form $\Gamma \vdash M : \sigma$ where Γ is a sequence of types. Types are preserved through rewriting

(we refer to [1] for details). Since we are interested here in strong normalization, we avoid empty work steps by adopting the following restriction: in an (Env) step, one of the antecedents must be a nonzero step reduction.

We define by induction on *types* sets $C(\sigma)$ ("C" for "computable") of closures which are subsets of the set $SN(\sigma)$ of strongly normalizable closures of type σ, and are stable under the operations of the syntax, so that it will become easy then to carry an induction on *terms*.

$$C(\kappa) = \{u \in SN(\kappa) \mid u \xrightarrow{*}_l a \text{ for some constant } a\},$$

$$C(\sigma \Rightarrow \tau) = \{u \in SN(\sigma \Rightarrow \tau) \mid u \xrightarrow{*}_l \lambda P[\nu] \text{ and } \forall v \in C(\sigma)\ P[v \cdot \nu] \in C(\tau)\}.$$

Remark 4.1. We have not yet proved that normal forms are constants or functional closures nor that the lazy strategy reaches terms of this form. All these additional properties will be proved together with the strong normalization property.

One readily observes that

$$u \in SN(\sigma),\ v \in C(\sigma),\ u \to_l v \Rightarrow u \in C(\sigma).$$

The folklore of computability is to show now that any closure is computable, which is clear for constants *if, as we assumed, constants have basic types.* (One may cope with other kinds of constants with more complicated definitions of computability.) The following lemma deals with the nonconstant closures.

Lemma 4.1. *Let M, τ, u_1, $\sigma_1, \ldots, u_n$, σ_n be as in the second typing rule for closures. If $u_i \in C(\sigma_i)$ for all i, then $M[u_1; \cdots; u_n] \in C(\tau)$.*

Proof. By induction on *terms*. If M is a variable i, then the closure lazily reduces to u_i. We only have to check that $i[u_1; \cdots; u_n]$ is strongly normalizable. A reduction path from $i[u_1; \cdots; u_n]$ must eventually apply (Access) since an infinite (Env) path would contradict that u_i is strongly normalizable for some i. So suppose the reduction path begins with $i[u_1; \cdots; u_n] \xrightarrow{*} i[u_1'; \cdots; u_n'] \to u_i'$. Then u_i' is a reduct of u_i, and is thus strongly normalizable. The case where M is an abstraction is similar. The final case is application. We infer by induction $M\,N[\rho] \to_l P[N[\rho] \cdot \nu]$ and $P[N[\rho] \cdot \nu] \in C(\tau)$. We are left to show that $M\,N[\rho]$ is strongly normalizable. Suppose, as for the variable case, that a reduction path from $M\,N[\rho]$ starts with $M\,N[\rho] \xrightarrow{*} M\,N[\rho'] \to P'[N[\rho'] \cdot \nu']$. Then by combining the antecedents we get $M[\rho] \xrightarrow{*} \lambda P'[\nu']$. But we also have by induction $M[\rho] \xrightarrow{*}_l \lambda P[\nu]$. Thus $P \equiv P'$, and moreover, by Lemma 2.2, $\nu \xrightarrow{*} \nu'$. Hence $P'[N[\rho'] \cdot \nu']$ is a reduct of $P[N[\rho] \cdot \nu]$, which entails that $P'[N[\rho'] \cdot \nu']$ is strongly normalizable. One easily concludes that there cannot be any infinite reduction path from $M\,N[\rho]$. □

The following statement collects what we have proved altogether.

Theorem 4.2. *The simply typed calculus of closures has the strong normalization property. Normal forms of closures of basic types are constants, and the lazy strategy*

always reaches them. Normal forms of terms of functional types are functional closures, and the lazy strategy terminates with a functional closure (which rewrites to the normal form by (Env)).

We could go on and prove strong normalization for more powerful type systems, beginning with second-order types à la Girard. We refer to [13] where this is done in a framework restricted to the eager strategy.

5. A strong calculus

In this section we briefly hint at an extension of the calculus of closures allowing to mimick the full β-reduction. We refer the reader to further publications [1, 8] which develop the syntactic theory of such strong calculi. We shall mainly insist on the categorical significance of the variant presented here.

We shall first give a precise definition of the substitution in De Bruijn's λ-calculus. In this definition, as in the calculus of closures presented in Section 2, the emphasis is on simultaneous substitution of all free variables, rather than substituting some variable with a term. The definition of simultaneous substitution in De Bruijn's calculus raises a subtle issue. Since all free variables have to be replaced simultaneously, one needs to know how many they are. De Bruijn came around the difficulty by defining infinite substitutions $M\{M_1/1, \ldots, M_n/n, \ldots\}$. Here is, *mutatis mutandis*, the definition originally proposed by De Bruijn [4]. First β-reduction is:

$$(\lambda a)b \rightarrow_{beta} a\{b/1, 1/2, \ldots, \mathrm{n}/\mathrm{n}+1, \ldots\}$$

where the metalevel substitution $\{\ldots\}$ is defined inductively by using the rules:

$$\mathrm{n}\{a_1/1, \ldots, a_n/\mathrm{n}, \ldots\} = a_n,$$

$$\frac{a\{a_1/1, \ldots, a_n/\mathrm{n}, \ldots\} = a' \quad b\{a_1/1, \ldots, a_n/\mathrm{n}, \ldots\} = b'}{(ab)\{a_1/1, \ldots, a_n/\mathrm{n}, \ldots\} = a'b'},$$

$$\frac{a_i\{2/1, \ldots, \mathrm{n}+1/\mathrm{n}, \ldots\} = a_i' \quad a\{1/1, a_1'/2, \ldots, a_n'/\mathrm{n}+1, \ldots\} = a'}{(\lambda a)\{a_1/1, \ldots, a_n/\mathrm{n}, \ldots\} = \lambda a'}.$$

The case of abstraction is rather complex, because variable captures must be avoided. The reader unfamiliar with De Bruijn's notation should try to reduce the translation of, say $\lambda y.(\lambda x.\lambda y.x)y$.

A way to "finitize" this definition, and to turn it into a system where substitution is explicit, is to switch to a calculus with arities. Arities will allow to control the number of free variables. We present below such a calculus. Here is the (abstract) syntax of our proposed strong calculus with arities (arities are numbers, and appear as superscripts).

$$\textit{Terms } A ::= m^n \text{ (where } m \leq n) \mid (A^n B^n)^n \mid (\lambda A^{n+1})^n \mid (A^m[\sigma^n])^n$$

where σ is a list of length m whose elements are all terms of sort n (we use the same syntax for lists as in the calculus of closures). The above grammar generates

the closed term algebra of the multisorted (polymorphic and dependent) signature whose sorts are nonnegative integers, and whose operators are

- m^n: n,
- application: $n \times n \to n$,
- abstraction: $n+1 \to n$,
- closure:

$$m \times \underbrace{n \times \cdots \times n}_{m \text{ times}} \to n.$$

Notice that now, in contrast to the weak calculus of Section 2, closures are just ordinary terms, and can in particular appear under λ's.

Here are the basic rewriting rules, which suffice to carry out the substitution process.

Beta $(\lambda A)B^n \to A[B; 1^n; \cdots; n^n]$,

Var $i^n[A_1; \cdots; A_n] \to A_i$,

App $AB[\sigma] \to (A[\sigma])(B[\sigma])$,

Abs $\lambda A^{n+1}[\sigma^m] \to \lambda(A[1^{m+1} \cdot (\sigma \circ [2^{m+1}; \cdots; (m+1)^{m+1}])])$.

In (Abs), we mean by $\sigma \circ \sigma'$ an abbreviation of $A_1[\sigma']; \cdots; A_n[\sigma']$, where $\sigma = A_1; \cdots; A_n$.

The last three rules form a subsystem which we call Dist. One may show that the system (Beta)+Dist is confluent on closed terms. One may turn it into a locally confluent (and, we believe, confluent) system by adding the two following rules:

Ass $(A[\sigma])[\sigma'] \to A[\sigma \circ \sigma']$,

Id $A^n[1^n; \cdots; n^n] \to A$.

We shall not insist further here on the syntactic theory of this calculus. We refer to [1, 8], where a friendlier syntax is proposed (arities, and "dot, dot, dot" notations are avoided by the use of further explicit operations on substitutions). The reason why we insist here on presenting the variant with arities is that it corresponds semantically to a notion of closed multicategory, which we briefly present now.

The well-known correspondence between λ-calculus and cartesian closed categories provides a deep foundation for the explicit treatment of substitution. Indeed the basic paradigm of categorical logic is substitution-as-composition. Clearly, composition is *the* primitive operation in category theory!

But, at least as presented in [16, 7], this correspondence, strictly taken, involves explicit pairs in the λ-calculus. It happens that a slightly different categorical setting allows for a correspondence with the simply typed λ-calculus, without a need of extending it with products. This setting, long known to Lambek [14] under the name of *multicategories*, and to Bénabou under the name of *catégories avec déduction* [3], has been used as early as 1977 by A. Obtulowicz [20] under the name of *Church algebraic theories* to yield such a correspondence (in an untyped setting). Recently

the technique has been reinvented at a syntactic level by H. Yokouchi and T. Hikita [23], who came up with a system which is very similar to the calculus presented above. The basic observation is that cartesian categories oblige to mix two uses of products: one for coding variable access, the other for modelling the explicit products. The point of explicit substitutions is to preserve the separation by resorting to the division between the sort of terms and the sort of substitutions.

In multicategories, arrows have sequences of objects as domains, and identity morphisms are not primitive: instead n-ary projections are primitive, and can serve to interpret variables without further encoding. Actually Lambek called multicategories even more general structures, corresponding to logical systems without weakening, like linear logic (an up-to-date discussion of these notions can be found in [15]). More formally, we shall understand here as a multicategory a structure with objects $A, B, \ldots$ and arrows $f: A_1, \ldots, A_n \to B$, where the following operations are available.

- For each sequence $A_1, \ldots, A_n$ and each $i \leqslant n$, a distinguished arrow $i: A_1, \ldots, A_n \to A_i$
- A composition operation which takes $f_1: A_1, \ldots, A_m \to B_1, \ldots, f_n: A_1, \ldots, A_m \to B_n$, $g: B_1, \ldots, B_n \to C$ and yields an arrow $g[f_1; \cdots; f_n]: A_1, \ldots, A_m \to C$. These operations have to satisfy the rules (Ass), (Var) and (Id).

Next we need a category-theoretic notion of closure under function spaces. We shall call a multicategory *closed* if for any pair of objects A, B there exists an object $A \Rightarrow B$ s.t. for any sequence of objects $C_1, \ldots, C_n$ there exist bijections λ mapping $f: C_1, \ldots, C_n, A \to B$ into $\lambda(f): C_1, \ldots, C_n \to A \Rightarrow B$, which are natural, i.e. such that the rule (Abs) is satisfied. This definition presents a slight mismatch with the syntax of the calculus presented above: there is no direct notion of binary application. It can be encoded as follows: let $App: A \Rightarrow B, A \to B$ be the unique arrow such that $\lambda(App) = 1: A \Rightarrow B \to A \Rightarrow B$. Then, for $f: C_1, \ldots, C_n \to A \Rightarrow B$, $g: C_1, \ldots, C_n \to A$, set $fg = App[f; g]$. (Inversely, App can be coded from binary application by $12: A \Rightarrow B, A \to B$.) Notice that now (App) becomes an instance or (Ass). Satisfaction of (Beta) is equivalent to the assumption $\lambda^{-1} \circ \lambda = id$.

Notice finally that the axiomatization of closed multicategories corresponds to the system Dist+(Ass)+(Id)+(Beta)+(Eta), where

Eta $\quad \lambda(21) \to 1$.

In order to stick to the β-rule only, one may relax the definition of closed multicategories to a notion of weakly closed categories, where the maps λ are only assumed to be retractions.

Acknowledgment

I wish to thank Yves Lafont, whose simple proof of termination of call-by-value evaluation prompted this study. I owe very much to the coauthors of [1, 8] Martin

Abadi, Luca Cardelli, Thérèse Hardin and Jean-Jacques Lévy. I also wish to thank Guy Cousineau and Gérard Huet for constant support.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien and J.-J. Lévy, Explicit substitutions, in: *Proc. Conf. on Principles of Programming Languages* 90, San Francisco (1990).

[2] S. Abramsky, The lazy λ-calculus, in: D. Turner, ed., *Declarative Programming* (Addison-Wesley, Reading, MA, to appear).

[3] J. Bénabou, Catégories et logiques faibles, Tagungsbericht 30, Mathematisches Forschungsinstitut Oberwolfach, 1973.

[4] N. de Bruijn, Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem, *Indag Math.* **34** (5) (1972) 381-392.

[5] L. Cardelli, The Amber Machine, in: G. Cousineau, P.-L. Curien and B. Robinet eds., *Combinators and Functional Programming Languages*, Lecture Notes in Computer Science **242** (Springer, Berlin, 1986).

[6] G. Cousineau, P.-L. Curien and M. Mauny, The Categorical Abstract Machine, *Sci. Comput. Programming* **8** (1987) 173-202.

[7] P.-L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming* (Pitman, London, 1986).

[8] T. Hardin and J.-J. Lévy, A confluent calculus of explicit substitutions, France-Japan Artificial Intelligence and Computer Science Symposium, Izer, 1989.

[9] J. Fairbairn and S. Wray, A simple, lazy, abstract machine to execute supercombinators, in: Lecture Notes in Computer Science **274** (Springer, Berlin, 1987).

[10] T. Hardin, Confluence results for the pure strong C.C.L. lambda-calculi as subsystems of C.C.L., *Theoret. Comput. Sci.* **65** (1989) 291-342.

[11] J.-L. Krivine, Unpublished notes.

[12] Y. Lafont, Logiques, catégories et machines, Thèse de Doctorat, Université Paris VII, 1987.

[13] Y. Lafont, Reducibility and evaluation, Seminar notes, 1988.

[14] J. Lambek, Deductive Systems and Categories II: Standard Constructions and Closed Categories, in: Lecture Notes in Mathematics **86** (Springer, Berlin, 1969).

[15] J. Lambek, Multicategories revisited, in: *Proc. A.M.S. Conf. on Categories in Computer Science and Logic, Boulder 1987*, Contemporary Mathematics **92** (1989).

[16] J. Lambek and P.J. Scott, *Introduction to Higher Order Categorical Logic* (Cambridge Univ. Press, Cambridge, UK, 1986).

[17] P.J. Landin, The mechanical evaluation of expressions, *Computing J.* **6** (1964) 308-320.

[18] M. Mauny, Compilation des langages fonctionnels dans les combinateurs catégoriques; Application au langage ML, Thèse de Troisième Cycle, Université Paris VII, 1985.

[19] J.-J. Lévy, Réductions correctes et optimales dans le lambda-calcul, Thèse de Doctorat d'Etat, Université Paris VII, 1978.

[20] A. Obtulowicz, Categorical, Functorial and Algebraic Aspects of the Type-free Lambda-calculus, in: *Universal Algebra and Applications*, Banach Center Publications **9** (Warsaw, 1982).

[21] C.-H. Luke Ong, Fully abstract models of the lazy λ-calculus, in: *Proc. 29th Conf. on Foundations of Computer Science, 1988* (Computer Society Press, Rockville, MD, 1988).

[22] G.D. Plotkin, Call-by-name, call-by-value and the λ-calculus, *Theoret. Comput. Sci.* **1** (1975) 125-159.

[23] H. Yokouchi and T. Hikita, A rewriting system for categorical combinators with multiple arguments, Preprint, 1988.