HAL
open science

Explaining the lazy Krivine machine using explicit substitution and addresses

Frederic Lang

To cite this version:

Frederic Lang. Explaining the lazy Krivine machine using explicit substitution and addresses. Higher-Order and Symbolic Computation, 2007. inria-00198756

HAL Id: inria-00198756

https://inria.hal.science/inria-00198756v1

Submitted on 17 Dec 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Explaining the lazy Krivine machine using explicit substitution and addresses

Frédéric Lang

INRIA Rhône-Alpes, 655 avenue de l'Europe, 38 330 Montbonnot, France.
Frederic.Lang@inrialpes.fr

Abstract. In a previous paper, Benaissa, Lescanne, and Rose, have extended the weak lambda-calculus of explicit substitution $\lambda\sigma_w$ with addresses, so that it gives an account of the sharing implemented by lazy functional language interpreters. We show in this paper that their calculus, called $\lambda\sigma_w^a$, fits well to the lazy Krivine machine, which describes the core of a lazy (call-by-need) functional programming language implementation. The lazy Krivine machine implements term evaluation sharing, that is essential for efficiency of such languages. The originality of our proof is that it gives a very detailed account of the implemented strategy.

1 Introduction

Functional language designers generally agree that, whereas the λ-calculus — or the lazy λ-calculus [2] as regards more specifically normal order reduction — is a useful framework to deal with program denotations, it is not fully appropriate to describing their operational semantics. Indeed, λ-calculus operational semantics suggest that each occurrence of a variable is *immediately* substituted by a *copy of the argument* after each redex contraction. This call-by-name strategy would be very inefficient in practice, particularly in the context of lazy functional programming languages.

Instead of reasoning directly in the λ-calculus, functional language implementations are generally based on *abstract machines*, focusing on a particular reduction strategy at a fine level of detail by using appropriate data structures. As regards lazy functional languages, following the seminal work of Wadsworth [26] on graph reduction, numerous abstract machines have been proposed, such as the lazy Secd [14], the G-machine [15], Tim [11], the lazy Krivine machine [7, 24], the Stg-machine [21], and Mark3 [25]. Most of these machines implement stacks and *closures* to avoid immediate argument substitutions, and *heap memories* to avoid argument copies (*sharing*). These machine's strategies, although slightly different by many aspects, are generally known as *call-by-need*. Yet, they may be difficult to understand or compare in terms of space and time complexity, which advocates for a common, more abstract and more general framework to deal with strategies at a sufficiently fine level of detail.

In the early 90's, *explicit substitution calculi* [1] have been proposed as such a framework. Hardin et al. [13] have used the $\lambda\sigma_w$ calculus [8, 9] to establish

the correspondence between steps of several abstract machines and derivations in $\lambda\sigma_w$. However, explicit substitution calculi still do not address sharing, which results in one abstract machine transition leading to an unbounded number of transitions in the calculus.

To keep track of sharing, Benaissa, Lescanne, and Rose [6] have proposed a calculus named $\lambda\sigma_w^a$, an extension of $\lambda\sigma_w$ with global term annotations called *addresses*, accompanied by a specific rewriting relation, named *simultaneous rewriting*, to model in a single step the computation performed on all shared instances. As a general rewriting system, $\lambda\sigma_w^a$ is not attached to a specific strategy: strict and lazy strategies have been defined on top of the calculus, independently of the rewriting rules.

We believe that $\lambda\sigma_w^a$ provides the good granularity of computation steps to reason about abstract machines, especially the lazy ones. As an illustration of this claim, we present in this paper an original proof of correctness of the lazy Krivine machine. This proof explains the semantics of the machine by establishing the precise correspondence between the states of the machine and the terms of the calculus. We show in particular that each computation step of the machine roughly corresponds to one reduction step in the calculus.

Plan of the paper. Section 2 presents the weak λ-calculus of explicit substitution and addresses $\lambda\sigma_w^a$, and the associated lazy strategies. Section 3 presents the lazy Krivine machine. Section 4 details a proof that the lazy Krivine machine implements a particular variant of call-by-need. We relate this work with other works in Section 5, and then conclude in Section 6.

2 The calculus $\lambda\sigma_w^a$

In this section, we present the weak λ-calculus of explicit substitution and addresses $\lambda\sigma_w^a$, proposed by Benaissa, Lescanne, and Rose [6].

2.1 Syntax

The $\lambda\sigma_w^a$ calculus is inspired by the $\lambda\sigma_w$ explicit substitution calculus [9], which is a (first-order) rewriting system that simulates weak β-reduction using a few simple rules. More precisely, following ideas initially introduced by Rose [23], $\lambda\sigma_w^a$ is $\lambda\sigma_w$ extended with so-called *addresses*, *i.e.*, term annotations that can be understood as memory locations, used to keep track of sharing.

The $\lambda\sigma_w^a$-terms are defined upon $\lambda\sigma_w^a$*-preterms*, defined recursively by the rules of Figure 1. There are two main categories of preterms, namely *code* and *evaluation contexts*. Code consists of pure, non-addressed, λ-terms in de Bruijn notation [10]. Evaluation contexts are structures enabling code evaluation: de Bruijn index $\underline{n}$, *closure* $M[s]$, made of code M and substitution s associating an evaluation context to each variable occurring free in M, or *evaluation context application* $U\,V$. All evaluation contexts are annotated with *addresses*, written

$$
\begin{aligned}
T, U, V &::= E^a \mid \bot && \text{(Addressed)}\\
E, F &::= M[s] \mid UV \mid \underline{n} && \text{(Evaluation Context)}\\
M, N &::= \lambda M \mid MN \mid \underline{n} && \text{(Code)}\\
s, t &::= \mathrm{id} \mid U \cdot s && \text{(Substitution)}
\end{aligned}
$$

where a, b, c range over an infinite set $\mathbf{A}$ of *addresses*.

Fig. 1. $\lambda\sigma_w^a$ syntax.

$a, b, c, \ldots$ Starting the evaluation of term M consists in creating an evaluation context of the form $M[\mathrm{id}]^a$ where id is the empty substitution.

One can see a $\lambda\sigma_w^a$-preterm as a directed acyclic graph, the nodes of which are term symbols, and such that two subterms with the same address represent sharing at this address. For sharing consistency reasons, pre-subterms with the same address must be identical. This so-called *admissibility* condition is formally defined as follows:

Definition 1 ($\lambda\sigma_w^a$-terms).

1. *The set of addresses occurring in a $\lambda\sigma_w^a$-preterm U (resp. an evaluation context E, a substitution s, a context C) is denoted by* $\mathrm{addr}(U)$ *(resp.* $\mathrm{addr}(E)$*,* $\mathrm{addr}(s)$*,* $\mathrm{addr}(C)$*) and defined in an obvious way.*
2. *The* subterms of U at a*, written $U@a$, is the set $\{E_1^a, \ldots, E_n^a\}$ for which an n-ary context C exists such that $U = C\{E_1^a, \ldots, E_n^a\}$ and $a \notin \mathrm{addr}(C)$.*
3. *A $\lambda\sigma_w^a$-preterm U is* admissible *(or is a $\lambda\sigma_w^a$-term) whenever for all $a \in \mathrm{addr}(U)$, $U@a = \{E^a\}$ and $a \notin \mathrm{addr}(E)$.*

In this paper, we only consider finite terms. By the admissibility condition, this implies that address a may not occur in proper subterms of E^a. In the following, we will only deal with admissible terms. Thus, for notation convenience, we will write $U@a$ to denote the term located at address a in U, provided $a \in \mathrm{addr}(U)$.

2.2 Dynamics

Two subterms are shared instances if they have the same address. Therefore, term evaluation sharing is modeled by rewriting all terms with the same address in the same step to the same evaluation context, called *simultaneous rewriting*.

$\lambda\sigma_w^a$-rewriting rules are given in Figure 2. They are pairs of admissible $\lambda\sigma_w^a$-terms with first-order variables. Note that:

- Rule (B_w) differs from usual β-contraction, and corresponds to weak β-contraction, in the sense that redexes are not reduced in the scope of a λ-abstraction.
- Rule (App) distributes substitution s to both sides of the application, which creates two closures located at new addresses b and c.

$$
\begin{aligned}
((\lambda M)[s]^b\, U)^a &\to M[U \cdot s]^a && && (\mathrm{B}_w)\\
(MN)[s]^a &\to (M[s]^b\, N[s]^c)^a && b, c \text{ fresh} && (\mathrm{App})\\
\underline{0}\,[E^b \cdot s]^a &\to E^b && && (\mathrm{FVarG})\\
\underline{0}\,[E^b \cdot s]^a &\to E^a && && (\mathrm{FVarE})\\
\underline{n+1}\,[U \cdot s]^a &\to \underline{n}\,[s]^a && && (\mathrm{RVar})\\
\underline{n}\,[\mathrm{id}]^a &\to \underline{n}^a && && (\mathrm{VarId})\\
M[s]^a &\to M[s|_{\mathrm{fi}(M)}]^a && && (\mathrm{Collect})
\end{aligned}
$$

where *environment trimming* is defined by $s|_I = s|_I^0$ where

$$
\begin{aligned}
\mathrm{id}|_I^i &= \mathrm{id}\\
(U \cdot s)|_I^i &= \begin{cases} U \cdot s|_I^{i+1} & \text{if } i \in I\\ \bot \cdot s|_I^{i+1} & \text{otherwise}\end{cases}
\end{aligned}
$$

and the *free indices* of a term M is the set $\mathrm{fi}(M) = \mathrm{fi}_0(M)$, fi_i being defined as follows:

$$
\begin{aligned}
\mathrm{fi}_i(M\,N) &= \mathrm{fi}_i(M) \cup \mathrm{fi}_i(N)\\
\mathrm{fi}_i(\lambda M) &= \mathrm{fi}_{i+1}(M)\\
\mathrm{fi}_i(\underline{n}) &= \begin{cases} \{\,\underline{n-i}\,\} & \text{if } n \geq i\\ \emptyset & \text{if } n < i\end{cases}
\end{aligned}
$$

Fig. 2. $\lambda\sigma_w^a$ reduction rules.

- Rules (FVarG) and (FVarE) model access to an evaluation context. The reason for introducing two rules is discussed in [6]. It roughly corresponds to the possibility of changing a term location by pointer indirection (FVarG) versus root copy (FVarE)[1].
- Rule (VarId) describes the end of computation when the environment reaches a free variable. This rule can be ignored if one considers only closed terms, as will be the case in the current paper.
- At last, rule (Collect) describes environment *trimming*. It removes from substitutions the terms bound to variables that do not occur in the code. We will use a slightly modified (and specialized) version of (Collect):

  $$\underline{0}\,[U \cdot s]^a \to \underline{0}\,[U \cdot \mathrm{id}]^a \qquad (\text{Collect'})$$

  The reason for using a specialized rule is related to the correctness proof presented in Section 4 and will be discussed in the conclusion.

In the rewriting semantics, addresses can be considered as a special kind of variables, with the convention that, unlike classical first-order variables, addresses that occur in the right hand side of a rule do not necessarily occur in the

[1] E and G stand respectively for *Environment* and *Graph*, since environment machines usually implement root copy whereas graph reducers are based on pointer indirections.

left hand side. At run-time, such addresses must be chosen *fresh*, *i.e.*, must be assigned values that do not belong to the term the rule is applied to.

Definition 2 (Simultaneous rewriting).

1. *The* replacement *of all the subterms at address a in U by V is denoted by $U\{a := V\}$ and defined in Figure 3.*

$$
\begin{array}{rcll}
E^a\{a := V\} & = & V & \\
\bot\{a := V\} & = & \bot & \\
M[s]^b\{a := V\} & = & M[s\{a := V\}]^b & \text{if } a \neq b \\
(U_1\, U_2)^b\{a := V\} & = & (U_1\{a := V\}\, U_2\{a := V\})^b & \text{if } a \neq b \\
\underline{n}^b\{a := V\} & = & \underline{n}^b & \text{if } a \neq b \\
\\
\mathrm{id}\{a := V\} & = & \mathrm{id} & \\
(U \cdot s)\{a := V\} & = & U\{a := V\} \cdot s\{a := V\} &
\end{array}
$$

Fig. 3. Replacement

2. *Given a rule $U \to V$ and a term W such that there is a substitution σ satisfying $W@a = \sigma(U)$, the result of (*simultaneously*) rewriting W at address a following rule $U \to V$ is $W' = W\{a := \sigma(V)\}$, written $W \overset{a}{\mapsto} W'$, where fresh addresses in V are chosen as addresses that do not occur in W.*

Notice that, since a replacement occurs at any place the address of the rewritten term occurs, and due to the use of fresh addresses, simultaneous rewriting preserves term admissibility [6].

Example 1. The term $U = (\lambda\, \underline{0}\, \underline{0})((\lambda\, \underline{0})\lambda\, \underline{0})[\mathrm{id}]^a$ (corresponding to the term $(\lambda x.x\, x)((\lambda y.y)\lambda z.z)[\mathrm{id}]^a$ in named notation) may reduce as follows:

$$
\begin{array}{lr}
U \overset{a}{\mapsto} ((\lambda\, \underline{0}\, \underline{0})[\mathrm{id}]^b((\lambda\, \underline{0})\lambda\, \underline{0})[\mathrm{id}]^c)^a & \text{(App)} \\
\overset{a}{\mapsto} (\underline{0}\, \underline{0})[((\lambda\, \underline{0})\lambda\, \underline{0})[\mathrm{id}]^c \cdot \mathrm{id}]^a & (\mathrm{B}_w) \\
\mapsto^* (\underline{0}\,[((\lambda\, \underline{0})[\mathrm{id}]^f(\lambda\, \underline{0})[\mathrm{id}]^g)^c \cdot \mathrm{id}]^d\, \underline{0}\,[((\lambda\, \underline{0})[\mathrm{id}]^f(\lambda\, \underline{0})[\mathrm{id}]^g)^c \cdot \mathrm{id}]^e)^a & \text{(App)} \\
\overset{c}{\mapsto} (\underline{0}\,[\underline{0}\,[(\lambda\, \underline{0})[\mathrm{id}]^g \cdot \mathrm{id}]^c \cdot \mathrm{id}]^d\, \underline{0}\,[\underline{0}\,[(\lambda\, \underline{0})[\mathrm{id}]^g \cdot \mathrm{id}]^c \cdot \mathrm{id}]^e)^a & (\mathrm{B}_w) \\
\overset{c}{\mapsto} (\underline{0}\,[(\lambda\, \underline{0})[\mathrm{id}]^c \cdot \mathrm{id}]^d\, \underline{0}\,[(\lambda\, \underline{0})[\mathrm{id}]^c \cdot \mathrm{id}]^e)^a & \text{(FVarE)} \\
\mapsto^* (\lambda\, \underline{0})[\mathrm{id}]^a &
\end{array}
$$

Sharing shows up in the third line of the derivation: the subterm rooted at address c is shared by both sides of the topmost evaluation context application. In the two next steps, both occurrences of this shared evaluation context are reduced at the same time.

Benaissa, Lescanne, and Rose have proven the soundness of $\lambda\sigma_w^a$ w.r.t. the calculus $\lambda\sigma_w$, which itself is sound w.r.t. the λ-calculus.

2.3 Call-by-need

The calculus $\lambda\sigma_w^a$ offers a very convenient framework to define reduction strategies. By defining restrictions on the reduction, it has been used to model most of the sequential and parallel strategies implemented in functional programming languages. In this paper, we will focus on call-by-need.

A strategy $\mathcal{S}$ is a relation that determines, at each step of a reduction, at which addresses the term can be reduced next, and since $\lambda\sigma_w^a$ is not an orthogonal system, which rule of (FVarE, FVarG) must be used. The definition of call-by-need can be done using rule components shown in Figure 4, to which restrictions given by $\mathcal{S}$ can be associated.

$$\frac{}{M[s]^a \vdash a}\ (\text{Scl}) \qquad \frac{s \vdash a}{M[s]^b \vdash a}\ (\text{Sub}) \qquad \frac{U \vdash a}{U \cdot s \vdash a}\ (\text{Hd})$$

$$\frac{}{(UV)^a \vdash a}\ (\text{Sap}) \qquad \frac{U \vdash a}{(UV)^b \vdash a}\ (\text{Lap})$$

Fig. 4. Basic components for the definition of call-by-need.

Definition 3. *We write $U \vdash_{\mathcal{S}} a$ if and only if $U \vdash a$ can be derived using the rules of Figure 4 following the restrictions in $\mathcal{S}$. We write $U \mapsto_{\mathcal{S}} V$ if and only if $U \vdash_{\mathcal{S}} a$ and $U \overset{a}{\mapsto} V$. We may also write $U \overset{a}{\mapsto}_{\mathcal{S}} V$ to name the address a where the reduction occurs following $\mathcal{S}$.*

Benaissa, Lescanne, and Rose [6] have identified two variants of call-by-need, called NeedE and NeedG, which differ essentially in the choice between (FVarG) and (FVarE). They are presented in Figure 5. Notice that there exist other slight variants of call-by-need, such as the one implemented in Stg [21] and Mark3 [25], the strategy of which uses both (FVarE) and (FVarG). In particular, (FVarG) models indirection elimination at the time of pushing closures on the stack.

Example 2. The derivation shown in Example 1 follows the NeedE strategy.

Note that the definition of strategies given here is relaxed for rules (Collect) and (Collect'), which can be applied at any time orthogonally to the other evaluation rules. Benaissa, Lescanne, and Rose [6] have shown the importance of this rule to avoid memory leaks.

3 The lazy Krivine machine

The lazy Krivine machine [7, 24] is an extension of the Krivine machine to perform call-by-need. It is quite similar in spirit to the Stg machine [21] and

Strategy	(FVar?)	(Scl)	(Sub)	(Hd)	(Sap)	(Lap)
NeedE	E	②	¬②	✓	①	¬①
NeedG	G	✓	×	×	①	¬①

✓: "use;" ×: "don't use;" ①: "use if U is a value;" ②: "use if $s = U \cdot s'$ and $M = \underline{0}$ and U is a value."

Fig. 5. Call-by-need strategies of $\lambda\sigma_w^a$.

Mark3 [25]. For esthetical reasons, we present the lazy Krivine machine in Stg like style, using both an *argument stack* and an *update stack*, unlike original presentations in which a single heterogeneous stack contains both closure addresses and update markers of the form $\mathbf{M}(a)$, where $\mathbf{M}$ is the marker symbol, and a an address[2].

Syntax.

$$
\begin{array}{rll}
S & ::= (M[e]\ ;\ s\ ;\ u\ ;\ h) & \text{(State)} \\
M, N & ::= \lambda M \ \mid\ MN \ \mid\ \underline{n} & \text{(Code)} \\
e, s & ::= a \cdot e \ \mid\ \epsilon & \text{(Env, Stack)} \\
u & ::= (s, a) \cdot u \ \mid\ \epsilon & \text{(Update Stack)} \\
h & ::= [\,] \ \mid\ h\big[a \mapsto M[e]\big] & \text{(Heap)} \\
 & a \text{ denotes } \textit{closure addresses} &
\end{array}
$$

Dynamics.

$$
\begin{array}{rll}
(MN[e]\ ;\ s\ ;\ u\ ;\ h) \to (M[e]\ ;\ a \cdot s\ ;\ u\ ;\ h\big[a \mapsto N[e]\big]) & a \text{ fresh} & \text{(App)} \\
(\lambda M[e]\ ;\ a \cdot s\ ;\ u\ ;\ h) \to (M[a \cdot e]\ ;\ s\ ;\ u\ ;\ h) & & \text{(Lam)} \\
(\underline{n+1}\,[a \cdot e]\ ;\ s\ ;\ u\ ;\ h) \to (\underline{n}\,[e]\ ;\ s\ ;\ u\ ;\ h) & & \text{(Skip)} \\
(\underline{0}\,[a \cdot e]\ ;\ s\ ;\ u\ ;\ h) \to (h(a)\ ;\ \epsilon\ ;\ (s, a) \cdot u\ ;\ h) & & \text{(Access)} \\
(\lambda M[e]\ ;\ \epsilon\ ;\ (s, a) \cdot u\ ;\ h) \to (\lambda M[e]\ ;\ s\ ;\ u\ ;\ h\big[a \mapsto \lambda M[e]\big]) & & \text{(Update)}
\end{array}
$$

Fig. 6. States and rules of the lazy Krivine machine

[2] Note that both forms are equivalent, since the unique stack can be recovered from the two stacks, using the simple bijective function δ defined as follows:

$$
\begin{array}{c}
\delta(a \cdot s, u) = a \cdot \delta(s, u) \\
\delta(\epsilon, (s, a) \cdot u) = \mathbf{M}(a) \cdot \delta(s, u) \\
\delta(\epsilon, \epsilon) = \epsilon
\end{array}
$$

States of the lazy Krivine machine have four components described in Figure 6, namely *code* with an *environment*, *argument stack*, *update stack*, and *heap*.

The dynamics of the machine are similar to Krivine's machine, except concerning the use of references in the heap, and details of memory management. In particular, rule (App) involves creation of a closure at a fresh location in the heap, rule (Access) involves dereferencing of some closure in the heap, and rule (Update) is a new administrative rule that updates some heap location with the weak head normal form of the argument stored at this location.

Note that in absence of explicit recursion operator, there are no cyclic dependencies between addresses: if $a \mapsto N[e]$ belongs to the heap, then a does not occur in $N[e]$. As a consequence, we do not address the "black-hole" problem (a closure of the form $\underline{0}\,[a{\cdot}e]$ where a is also the address of the closure), which occurs in abstract machines for λ-terms extended with an explicit recursion operator.

4 Proof of the lazy Krivine machine

Our purpose is to give a simple and formal proof of correctness of the lazy Krivine machine using $\lambda\sigma_w^a$, and show that it performs NeedE to reduce terms to weak head normal form (when one exists). To this end, we define an original *read-back* function to get from machine states into $\lambda\sigma_w^a$ terms. We then prove several technical lemmas that we use to demonstrate the soundness of the lazy Krivine machine with respect to the NeedE strategy.

Definition 4. *Let $\mathcal{A}$ be a countably infinite subset of $\mathbf{A}$ (the set of $\lambda\sigma_w^a$ addresses) whose elements are denoted by $\alpha, \alpha_0, \alpha_1, \ldots$ We assume that if a and b are distinct addresses of the lazy Krivine machine, then they are also distinct addresses of $\mathbf{A}$, but they do not belong to $\mathcal{A}$.*

The set $\mathcal{A}$ is a technicality that will be used to denote addresses that are implicit in the lazy Krivine machine but must be named in $\lambda\sigma_w^a$. For instance, each stack frame of the machine is implicitly located at some address, which is made explicit in $\lambda\sigma_w^a$ as the address of an evaluation context application.

Definition 5. *The read-back function ν from states of the lazy Krivine machine into $\lambda\sigma_w^a$ preterms is defined as follows:*

$$\nu(M[e]\ ;\ s\ ;\ u\ ;\ h) = \tau(M[\gamma(e, \varphi(h))]^\alpha, s, u, \varphi(h))$$

where $\alpha \in \mathcal{A}$ and:

- *φ is a function that unfolds the heap as a mapping from $\mathbf{A}$ to $\lambda\sigma_w^a$-terms[3], as follows:*

$$\varphi(h\big[a \mapsto N[e]\big]) = \varphi(h)\big[a \mapsto N[\gamma(e, \varphi(h))]^a\big]$$
$$\varphi([\,]) = [\,]$$

[3] Be aware that objects written h (heaps) are different from objects written ρ or $\varphi(h)$ (mappings) in that $h(a)$ denotes the machine closure located at a, whereas $\rho(a)$ denotes the corresponding $\lambda\sigma_w^a$ term.

– *γ reads back environments. It takes an environment and a mapping to $\lambda\sigma_w^a$ terms, and returns a $\lambda\sigma_w^a$ substitution, as follows:*

$$\gamma(a \cdot e, \rho) = \rho(a) \cdot \gamma(e, \rho)$$
$$\gamma(\epsilon, \rho) = \mathrm{id}$$

– *τ reads back stacks as an evaluation context application. It associates a single $\lambda\sigma_w^a$ term to a quadruple consisting of a $\lambda\sigma_w^a$ term, an argument stack, an update stack, and a mapping to $\lambda\sigma_w^a$ terms, as follows:*

$$\tau(E^b, a \cdot s, u, \rho) = \tau\big((E^b\, \rho(a))^\alpha, s, u, \rho\big) \text{ where } \alpha \in \mathcal{A}$$
$$\tau(E^b, \epsilon, (s, a) \cdot u, \rho) = \tau(\,\underline{0}\,[E^a \cdot \mathrm{id}]^b, s, u, \rho[a \mapsto E^a])$$
$$\tau(E^a, \epsilon, \epsilon, \rho) = E^a$$

– *Addresses in $\mathcal{A}$ are chosen such that each has at most one occurrence in $\nu(S)$ for any machine state S.*

In the following, we consider two $\lambda\sigma_w^a$ terms identical modulo any bijective renaming of addresses in $\mathcal{A}$ that permits to translate one term into the other.

Definition 6. *The replacement $\{a := U\}$ is extended to mappings from addresses to $\lambda\sigma_w^a$-terms as follows:*

$$\rho[b \mapsto V]\{a := U\} = \rho\{a := U\}[b \mapsto V\{a := U\}]$$
$$[\,]\{a := U\} = [\,]$$

In other words, for all addresses a, b, $\rho\{a := U\}(b) = \rho(b)\{a := U\}$.

Lemma 1 shows that reducing the leftmost evaluation context of an application following NeedE is indeed an application of NeedE on the whole term. Additionally, this reduction step may affect the rightmost evaluation context by side effect.

Lemma 1. *Let E^a be a term and ρ a mapping such that $\big(E^a\, \rho(a')\big)^c$ is admissible.*

$$E^a \overset{b}{\mapsto}_{\text{NeedE}} F^a \Rightarrow \big(E^a\, \rho(a')\big)^c \overset{b}{\mapsto}_{\text{NeedE}} \big(F^a\, \rho\{b := F^a@b\}(a')\big)^c.$$

Proof. From the definition of $\lambda\sigma_w^a$, and from the definition of NeedE, if $E^a \overset{b}{\mapsto}_{\text{NeedE}} F^a$ then

$$(E^a\, E'^{a'})^c \overset{b}{\mapsto}_{\text{NeedE}} (F^a\, E'^{a'}\{b := F^a@b\})^c.$$

Take $E'^{a'} = \rho(a')$ and observe that $\rho(a)\{b := E^b\} = \rho\{b := E^b\}(a)$.

Lemma 2 draws the relationship between NeedE reduction and the machine state structure (stacks and heaps). It shows that the heap captures all the necessary sharing without any need for updating the stacks.

Lemma 2. *Let E^a, s, u, ρ be respectively a $\lambda\sigma^a_w$-term, an argument stack, an update stack, and a mapping, such that $\tau(E^a, s, u, \rho)$ is admissible.*

$$E^a \overset{b}{\mapsto}_{\text{NEEDE}} F^a \Rightarrow \tau(E^a, s, u, \rho) \mapsto_{\text{NEEDE}} \tau(F^a, s, u, \rho\{b := F^a@b\}).$$

Proof. In the following, ρ' stands for $\rho\{b := F^a@b\}$. We proceed by induction on u.

- Assume u is empty. We proceed by induction on s. First assume s is empty. Then $\tau(E^a, \epsilon, \epsilon, \rho) = E^a \mapsto_{\text{NEEDE}} F^a = \tau(F^a, \epsilon, \epsilon, \rho')$ by hypothesis. Assume now that s is not empty. Then it may be written $a' \cdot s'$ and

  $$\tau(E^a, a' \cdot s', \epsilon, \rho) = \tau\big((E^a\, \rho(a'))^\alpha, s', \epsilon, \rho\big).$$

  By Lemma 1, $\big(E^a\, \rho(a')\big)^\alpha \overset{b}{\mapsto}_{\text{NEEDE}} \big(F^a\, \rho'(a')\big)^\alpha$. Moreover, the term $\big(F^a\, \rho'(a')\big)^\alpha$ is admissible because, by hypothesis, $\big(E^a\, \rho'(a')\big)^\alpha$ is a subterm of an admissible term, hence is admissible, and simultaneous reduction preserves admissibility. Therefore, Since F^a contains b, then it is clear that $\big(F^a\, \rho'(a')\big)^\alpha@b = F^a@b$. Hence, by the inductive hypothesis on s',

  $$\begin{aligned}\tau(E^a, a' \cdot s', \epsilon, \rho) &\mapsto_{\text{NEEDE}} \tau\big((F^a\, \rho'(a'))^\alpha, s', \epsilon, \rho'\big)\\ &= \tau(F^a, a' \cdot s', \epsilon, \rho').\end{aligned}$$

- Assume now u is not empty. We proceed by induction on s. First assume s is empty, and let $U = \tau(E^a, \epsilon, (a', s') \cdot u, \rho)$.

  $$\begin{aligned}U &= \tau(\underline{0}\,[E^{a'} \cdot \mathrm{id}]^a, s', u, \rho[a' \mapsto E^{a'}])\\ &\overset{b}{\mapsto}_{\text{NEEDE}} \tau(\underline{0}\,[F^{a'} \cdot id]^a, s', u, \rho[a' \mapsto E^{a'}]\{b := F^{a'}@b\})\\ &\qquad \text{by i.h. on } u \text{ and since } \underline{0}\,[E^{a'} \cdot \mathrm{id}]^a \overset{b}{\mapsto}_{\text{NEEDE}} \underline{0}\,[F^{a'} \cdot \mathrm{id}]^a\\ &= \tau(\underline{0}\,[F^{a'} \cdot id]^a, s', u, \rho'[a' \mapsto F^{a'}]) \text{ since } E^{a'}\{b := F^{a'}@b\} = F^{a'}\\ &= \tau(F^a, \epsilon, (a', s') \cdot u, \rho').\end{aligned}$$

  Assume now s is not empty *i.e.*, it can be written $a' \cdot s'$, and let U be $\tau(E^a, a' \cdot s', u, \rho)$.

  $$\begin{aligned}U &= \tau\big((E^a\, \rho(a'))^\alpha, s', u, \rho)\\ &\overset{b}{\mapsto}_{\text{NEEDE}} \tau\big((F^a\, \rho'(a'))^\alpha, s', u, \rho')\\ &\qquad \text{by Lemma 1, i.h. on } s', \text{ and since } \big(F^a\, \rho'(a')\big)^\alpha@b = F^a@b.\\ &= \tau(F^a, a' \cdot s, u, \rho')\end{aligned}$$

An immediate corollary follows, when the reduction occurs at the root a of the term E^a, and this address does not belong to the heap domain, which is the case of addresses belonging to the set $\mathcal{A}$. This is a result which will be very often used in the main proof since, by definition, the address assigned to the code component of a machine state by the read-back function ν indeed always belongs to $\mathcal{A}$.

Corollary 1. *Let E^a, s, u, ρ be respectively a $\lambda\sigma_w^a$-term, an argument stack, an update stack, and a mapping, such that $\tau(E^a, s, u, \rho)$ is admissible, and a does not occur in ρ.*

$$E^a \overset{a}{\mapsto}_{\textsc{Neede}} F^a \Rightarrow \tau(E^a, s, u, \rho) \mapsto_{\textsc{Neede}} \tau(F^a, s, u, \rho).$$

Lemmas 3 and 4 allow "garbage introduction" during reasoning, which is a useful way of accounting for allocated terms in the main proof. Lemma 3 operates at the machine state level, whereas Lemma 4 directly exploits the absence of cyclic address dependencies in $\lambda\sigma_w^a$ terms.

Lemma 3. *If a does not occur in $(M[e]\ ;\ s\ ;\ u\ ;\ h)$ then, for any $N[e']$, $\nu(M[e]\ ;\ s\ ;\ u\ ;\ h) = \nu(M[e]\ ;\ s\ ;\ u\ ;\ h\big[a \mapsto N[e']\big])$.*

Lemma 4. *If $M[\gamma(e,\rho)]^a$ is admissible, then $\gamma(e,\rho) = \gamma(e, \rho[a \mapsto U])$ for all U.*

Proof. Since $M[\gamma(e,\rho)]^a$ is admissible, a may not occur as a proper subaddress. Therefore, $\rho[a \mapsto U]$ is never applied to a, and for any other address b, $\rho(b) = \rho[a \mapsto U](b)$.

Theorem 1 is the main result of this paper: it shows that machine derivations preserve term admissibility via the read-back function, and correspond to Neede reductions. Moreover, each reduction step in the abstract machine corresponds to one reduction step in the $\lambda\sigma_w^a$ calculus.

Theorem 1. *Let M_0 be a pure term such that*

$$(M_0[\epsilon]\ ;\ \epsilon\ ;\ \epsilon\ ;\ [\,]) \to^* (M[e]\ ;\ s\ ;\ u\ ;\ h) \to (M'[e']\ ;\ s'\ ;\ u'\ ;\ h').$$

1. *$\nu(M'[e']\ ;\ s'\ ;\ u'\ ;\ h')$ is admissible, and*
2. *$\nu(M[e]\ ;\ s\ ;\ u\ ;\ h) \mapsto_{\textsc{Neede}} \nu(M'[e']\ ;\ s'\ ;\ u'\ ;\ h')$.*

Proof. Since $\nu(M_0[\epsilon]\ ;\ \epsilon\ ;\ \epsilon\ ;\ [\,])$ is admissible, the first part of this proof only consists in showing that if $\nu(M[e]\ ;\ s\ ;\ u\ ;\ h)$ is admissible, then so is $\nu(M'[e']\ ;\ s'\ ;\ u'\ ;\ h')$. However, this is implied by 2, since simultaneous rewriting preserves admissibility. Hence in the following we do not care about 1 anymore.

We then prove 2 by case analysis on each transition of the lazy Krivine machine. In the following, ρ stands for $\varphi(h)$ and ρ' for $\rho\big[a \mapsto N[\gamma(e,\rho)]^a\big]$.

– (App). Let $U = \nu(MN[e]\ ;\ s\ ;\ u\ ;\ h)$.

$$\begin{aligned}
U &= \nu(MN[e]\ ;\ s\ ;\ u\ ;\ h\big[a \mapsto N[e]\big]) \\
&\qquad \text{by Lemma 3, where } a \text{ does not occur anywhere else} \\
&= \tau\big((MN)[\gamma(e,\rho')]^{\alpha_0}, s, u, \rho'\big) \\
&\mapsto_{\textsc{Neede}} \tau\big((M[\gamma(e,\rho')]^{\alpha_1} N[\gamma(e,\rho')]^a)^{\alpha_0}, s, u, \rho'\big) \\
&\qquad \text{by Corollary 1, and since } a \text{ and } \alpha_1 \text{ are fresh} \\
&= \tau\big((M[\gamma(e,\rho')]^{\alpha_1}\ \rho'(a))^{\alpha_0}, s, u, \rho'\big) \\
&\qquad \text{since } \rho'(a) = N[\gamma(e,\rho)]^a = N[\gamma(e,\rho')]^a \text{ by Lemma 4} \\
&= \tau(M[\gamma(e,\rho')]^{\alpha_1}, a \cdot s, u, \rho') \quad \text{by definition of } \tau \\
&= \nu(M[e]\ ;\ a \cdot s\ ;\ u\ ;\ h\big[a \mapsto N[e]\big]) \quad \text{by definition of } \nu
\end{aligned}$$

– (Lam). Let $U = \nu(\lambda M[e]\ ;\ a \cdot s\ ;\ u\ ;\ h)$.

$$\begin{aligned}
U &= \tau\big((\lambda M)[\gamma(e,\rho)]^{\alpha_0}, a \cdot s, u, \rho\big) \\
&= \tau\big(((\lambda M)[\gamma(e,\rho)]^{\alpha_0}\ \rho(a))^{\alpha_1}, s, u, \rho\big) \\
&\mapsto_{\textsc{NeedE}} \tau(M[\rho(a) \cdot \gamma(e,\rho)]^{\alpha_1}, s, u, \rho) \quad \text{by Corollary 1} \\
&= \tau(M[\gamma(a \cdot e,\rho)]^{\alpha_1}, s, u, \rho) \\
&= \nu(M[a \cdot e]\ ;\ s\ ;\ u\ ;\ h)
\end{aligned}$$

– (Skip). Let $U = \nu(\underline{n+1}\,[a \cdot e]\ ;\ s\ ;\ u\ ;\ h)$.

$$\begin{aligned}
U &= \tau(\underline{n+1}\,[\gamma(a \cdot e,\rho)]^{\alpha}, s, u, \rho) \\
&= \tau(\underline{n+1}\,[\rho(a) \cdot \gamma(e,\rho)]^{\alpha}, s, u, \rho) \\
&\mapsto_{\textsc{NeedE}} \tau(\underline{n}\,[\gamma(e,\rho)]^{\alpha}, s, u, \rho) \quad \text{by Corollary 1} \\
&= \nu(\underline{n}\,[e]\ ;\ s\ ;\ u\ ;\ h)
\end{aligned}$$

– (Access). Let $U = \nu(\underline{0}\,[a \cdot e]\ ;\ s\ ;\ u\ ;\ h)$.

$$\begin{aligned}
U &= \tau(\underline{0}\,[\gamma(a \cdot e,\rho)]^{\alpha}, s, u, \rho) \\
&= \tau(\underline{0}\,[\rho(a) \cdot \gamma(e,\rho)]^{\alpha}, s, u, \rho) \\
&\mapsto \tau(\underline{0}\,[\rho(a) \cdot \mathrm{id}]^{\alpha}, s, u, \rho) \text{ by Corollary 1 and rule (Collect')} \\
&= \tau(\underline{0}\,[\rho(a) \cdot \mathrm{id}]^{\alpha}, s, u, \rho[a \mapsto \rho(a)]) \text{ since } \rho[a \mapsto \rho(a)] = \rho \\
&= \tau(N[\gamma(e',\rho)]^{\alpha}, \epsilon, (s,a) \cdot u, \rho) \\
&\qquad \text{since } h\,a = N[e'] \text{ implies } \rho(a) = N[\gamma(e',\rho)]^{a} \\
&= \nu(N[e']\ ;\ \epsilon\ ;\ (s,a) \cdot u\ ;\ h)
\end{aligned}$$

Notice that this step of the machine does not correspond to a step in NeedE, but to a (Collect') step, which is correct given the remark at the end of Section 2.3.

– (Update). In the following, ρ' stands for $\rho\big[a \mapsto \lambda M[\gamma(e,\rho)]^{a}\big]$. Let $U = \nu(\lambda M[e]\ ;\ \epsilon\ ;\ (s,a) \cdot u\ ;\ h)$.

$$\begin{aligned}
U &= \tau\big((\lambda M)[\gamma(e,\rho)]^{\alpha}, \epsilon, (s,a) \cdot u, \rho\big) \\
&= \tau(\underline{0}\,[\lambda M[\gamma(e,\rho)]^{a} \cdot \mathrm{id}]^{\alpha}, s, u, \rho') \\
&= \tau(\underline{0}\,[\lambda M[\gamma(e,\rho')]^{a} \cdot \mathrm{id}]^{\alpha}, s, u, \rho') \quad \text{by Lemma 4} \\
&\mapsto_{\textsc{NeedE}} \tau(\lambda M[\gamma(e,\rho')]^{\alpha}, s, u, \rho') \quad \text{by Corollary 1} \\
&= \nu(\lambda M[e]\ ;\ s\ ;\ u\ ;\ h\big[a \mapsto \lambda M[e]\big])
\end{aligned}$$

Corollary 2. *The lazy Krivine machine reduces closed terms to their weak head normal form following* NeedE.

Proof. Notice that the final states have the form $(\lambda M[e]\ ;\ \epsilon\ ;\ \epsilon\ ;\ h)$ and represent $\lambda\sigma_w^a$-terms of the form $(\lambda M)[s]^a$ *i.e.*, weak head normal forms. Notice that $(\underline{n}\,[\epsilon]\ ;\ s\ ;\ u\ ;\ h)$ is not a stopping state since this is not the representation of a closed term.

5 Related work

Correctness of an extension of the lazy Krivine machine for reduction of terms to full normal form, called KNP, has been proven by Crégut [7]. In this work, states of KNP were projected on $\lambda\sigma$-terms [1]. Hardin et al. [13] gave a similar proof for the lazy Krivine machine, using the weak λ-calculus with explicit substitution $\lambda\sigma_w$. Both proofs state that the lazy Krivine machine implements a strategy implementing normal order reduction, but they have no mean to express the amount of sharing performed by the machine.

Sestoft [25] also gave a proof of a variant of the lazy Krivine machine, using the natural semantics of Launchbury [19] for lazy evaluation. Similarly, Mountjoy [20] derived an STG-like abstract machine starting from a variant of Launchbury's natural semantics and has proven some properties of the derived machine, including correctness. These proofs give an account of the sharing performed by both machines. However, they use a big step semantics, which therefore does not have a precise notion of elementary computation step and does not permit to reason precisely in terms of space and time complexity. Moreover, the semantics combines at the same time the computation rules and the order in which they must be applied: it is therefore not as generic as $\lambda\sigma_w^a$ and thus could not allow comparisons between various implementations. In particular, variants of the call-by-need strategy require to redefine the semantics. Ariola and Felleisen [5] have defined a small step semantics of call-by-need, using a λ-calculus extended with a let construct, but here again, the calculus and its strategy can not be separated.

More recently, Ager, Biernacki, Danvy, and Midtgaard have addressed the problem of deriving abstract machines, including Krivine's machine, using continuation passing style transformations applied to evaluators and interpreters of the λ-calculus [3, 4]. This is a slightly different approach than the one we have chosen here, because it relates programs (interpreters and abstract machines) that implement the same reduction strategy. Moreover, so far this work does not address sharing.

Note that Benaissa, Lescanne, and Rose have also included in the calculus $\lambda\sigma_w^a$ operators to handle recursion and pattern matching. The proof presented in this paper has been extended in the PhD thesis of the author [16] to consider a more complete lazy machine, taking advantage of the full power of $\lambda\sigma_w^a$.

Originally proposed by Rose [22] in the setting of Combinatory Reduction Systems, addresses are one step beyond explicit substitution, in their aim of modeling closer to implementations. In [18], we have studied a framework for reasoning about implementations of object oriented languages, using addresses and explicit substitution. Additionally to sharing and cycles, which were already handled by Rose's work, this framework also allows reasoning about mutation of state, that is an essential part of object orientation. This work also provides a way to represent indirection pointers explicitly, which may be useful in some circumstances (see discussion in the conclusion below). We have integrated all features in a single formalism, namely Addressed Term Rewriting Systems, in [17].

6 Conclusion

We have illustrated that explicit substitution and addresses together are useful tools for reasoning about implementations at a rather detailed level. The calculus $\lambda\sigma_w^a$ gives a nice understanding of evaluation strategies. In particular, each reduction step of an abstract machine can be mapped onto a constant number of steps of the $\lambda\sigma_w^a$ calculus (1 in the case of the lazy Krivine machine). This makes $\lambda\sigma_w^a$ a good candidate for reasoning about space and time complexity of abstract machines.

Note however that in order to achieve the proof, we have had to slightly modify the $\lambda\sigma_w^a$ calculus, namely by introducing a specific rule named (Collect'). This rule was introduced to give an account — by using a degenerated form of closure — of an indirection to the current subterm under evaluation, implemented by a frame on top of the update stack. In the more general setting of Addressed Term Rewriting [17], we have proposed to represent indirection nodes explicitly, as it is useful to capture important details of implementations.

The lazy Krivine machine is probably the simplest implementation of lazy evaluation one can find. Nevertheless, it is realistic and very close to the core of the STG abstract machine [21], which has led to an efficient implementation of the Glasgow Haskell compiler. In fact, we believe that the same proof could be achieved for STG or MARK3 [25] without fundamental changes.

In the future, it would be interesting to study how $\lambda\sigma_w^a$ can be used instead of a particular abstract machine to establish the correctness of analysis algorithms. For instance, Gustavsson [12] proves correct a sharing analysis algorithms determining when updates can be safely omitted, taking the lazy Krivine machine as the execution model. Performing the proof on $\lambda\sigma_w^a$ while identifying minimal strategy conditions would have obvious benefits, since it would suffice to prove that a particular implementation satisfies the strategy restrictions in order to safely apply the proven algorithm.

Acknowledgements

Most of this work was done at *École Normale Supérieure de Lyon* between 1997 and 1998, and was presented in the PhD thesis of the author [16]. The author is grateful to Andrea Asperti, Michel Mauny, and Laurence Puel who reviewed the thesis and made useful comments, to René David and Dehlia Kesner for their active participation in the board of examiners, and to Zine El-Abidine Benaissa, Daniel Dougherty, Pierre Lescanne, and Kristoffer Rose for their encouragements about the current work. The author also warmly thanks anonymous referees of this article for the quality of their reviews and the pertinence of their remarks.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

2. S. Abramsky. The lazy lambda calculus. *Research Topics in Functional Programming*, pages 65–116, 1990.
3. M.S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. From Interpreter to Compiler and Virtual Machine: A Functional Derivation. Technical Report RS-03-14, Basic Research in Computer Science (BRICS), Denmark, 2003.
4. M.S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A Functional Correspondence between Evaluators and Abstract Machines. Technical Report RS-03-13, Basic Research in Computer Science (BRICS), Denmark, 2003.
5. Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.
6. Z. E.-A. Benaissa, P. Lescanne, and K. H. Rose. Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution. In *Proceedings of the Symposium on Programming Languages: Implementation, Logics and Programs PLILP'96*, number 1140 in LNCS, 1996.
7. P. Crégut. *Machines à environnement pour la réduction symbolique et l'évaluation partielle*. PhD thesis, Université de Paris 7, 1991.
8. P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.
9. P.-L. Curien, T. Hardin, and J.-J. Lévy. Weak and Strong Confluent Calculi of Explicit Substitutions. *Journal of the ACM*, 43(2), 1996.
10. N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. Koninkl. Nederl. Akademie van Wetenschappen*, 75(5):381–392, 1972.
11. J. Fairbairn and S. C. Wray. TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators. In G. Kahn, editor, *Functional Programming and Computer Architecture*, number 274 in LNCS, pages 34–45, Portland, Oregon, 1987.
12. J. Gustavsson. A Type Based Sharing Analysis for Update Avoidance and Optimisation, 1999. Thesis for the Licentiate Degree, Chalmers, Göteborg University.
13. T. Hardin, L. Maranget, and B. Pagano. Functional back-ends within the lambda-sigma calculus. In *Proceeding of the International Conference on Functional Programming ICFP'96*, 1996.
14. P. Henderson. *Functional Programming—Application and Implementation*. Prentice-Hall, 1980.
15. T. Johnsson. Efficient Compilation of Lazy Evaluation. *SIGPLAN Notices*, 19(6):58–69, June 1984.
16. F. Lang. *Modèles de la 'b-réduction pour les implantations*. PhD thesis, École Normale Supérieure de Lyon, December 1998.
17. F. Lang, D. J. Dougherty, P. Lescanne, and K. H. Rose. Addressed Term Rewriting Systems. Technical report, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 1999.
18. F. Lang, P. Lescanne, and L. Liquori. A framework for defining object-calculi. In *Proceedings of Formal Methods Europe FME'99*, 1999.
19. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Principles Of Programming Languages*, 1993.
20. J. Mountjoy. The spineless tagless G-machine, naturally. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming ICFP'99 (Baltimore, Maryland, United States)*, pages 163 – 173. ACM Press, 1998.
21. S. L. Peyton Jones. Implementing Lazy Functional Programming Languages on Stock Hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

22. K. H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, Dept. of Computer Science, Univ. of Copenhagen, Universitetsparken 1, DK-2100 København Ø, February 1996. DIKU report 96/1.
23. Kristoffer Høgsbro Rose. Explicit Cyclic Substitutions. In M. Rusinowitch and J.-L. Rémy, editors, *Proceedings of the 3rd International Workshop on Conditional Term Rewriting Systems CTRS'92*, number 656 in LNCS. Springer-Verlag, July 1992.
24. P. Sestoft. *Analysis and efficient implementation of functional programs.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1991.
25. P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3), May 1997.
26. C.P. Wadsworth. *Semantics and pragmatics of the lambda calculus.* PhD thesis, Oxford, 1971.