Open access • Proceedings Article • DOI:10.1145/3131851.3131855

Environments and the complexity of abstract machines — Source link

Beniamino Accattoli, Bruno Barras

Institutions: École Polytechnique

Published on: 09 Oct 2017 - Principles and Practice of Declarative Programming

Topics: Data structure and Abstract machine

Environments and the Complexity of Abstract Machines.

Beniamino Accattoli, Bruno Barras

Environments and the Complexity of Abstract Machines

Beniamino Accattoli
INRIA, UMR 7161, LIX, École Polytechnique
beniamino.accattoli@inria.fr

Bruno Barras
INRIA, UMR 7161, LIX, École Polytechnique
bruno.barras@inria.fr

ABSTRACT

Abstract machines for functional languages rely on the notion of environment, a data structure storing the previously encountered and delayed beta-redexes. This paper provides a close analysis of the different approaches to define and implement environments. There are two main styles. The most common one is to have many local environments, one for every piece of code in the data structures of the machine. A minority of works instead uses a single global environment. Up to now, the two approaches have been considered equivalent, in particular at the level of the complexity of the overhead: they have both been used to obtain bilinear bounds, that is, linear in the number of beta steps and in the size of the initial term.

We start by having a close look on global environments and how to implement them. Then we show that local environments admit implementations that are asymptotically faster than global environments, lowering the dependency from the size of the initial term from linear to logarithmic, thus improving the bounds in the literature. We then focus on a third style, split environments, that are in between local and global ones, and have the benefits of both. Finally, we provide a call-by-need machine with split environments for which we prove the new improved bounds on the overhead.

CCS CONCEPTS

• **Theory of computation → Lambda calculus**; **Abstract machines**; *Operational semantics*; • **Software and its engineering → Functional languages**;

KEYWORDS

λ-calculus, abstract machines, complexity, implementations, cost models, call-by-need, Coq

ACM Reference Format:
Beniamino Accattoli and Bruno Barras. 2017. Environments and the Complexity of Abstract Machines. In *Proceedings of PPDP'17, Namur, Belgium, October 9–11, 2017,* 13 pages.
https://doi.org/10.1145/3131851.3131855

This work is part of a wider research effort, the COCA HOLA project [1].

PPDP'17, October 9–11, 2017, Namur, Belgium

ACM ISBN 978-1-4503-5291-8/17/10…$15.00
https://doi.org/10.1145/3131851.3131855

1 INTRODUCTION

Functional programming languages and proof assistants are based on the λ-calculus, that in turn rests on a powerful computational rule, β-reduction. Its power is well expressed by a degeneracy known as *size explosion*: there are programs whose size can grow exponentially with the number of β-steps. In practice, then, β-reduction is never implemented as it is specified in the λ-calculus. The rule is decomposed, and the process of substitution on which it is based is delayed and performed in *micro steps* and *on-demand*: implementations replace only one variable occurrence at the time (micro steps) and only when the value of such an occurrence is needed for the evaluation to continue (on-demand).

Environments. Implementation schemas are called *abstract machines*, and they usually rely on a data structure called *environment*, storing the previously encountered and delayed β-redexes. Most of the literature on abstract machines relies on a style of environments that we like to call *local*, in which every piece of code in the machine is paired with its own environment, forming a *closure*—as in Krivine Abstract Machine [30], for instance. Other styles exist, however. A minority of works [3–5, 7, 9, 22, 24, 29, 38, 39] rather employs a single *global* environment, and the literature contains also an example of a mixed style due to Sestoft [39], that we like to call *split* environment. In this paper we study the features and the issues of these notions of environment with respect to their computation complexity, also discussing how they can be concretely implemented.

Roughly, local environments allow to avoid α-renaming, while global environments require α-renaming but allow for more sharing, and are essential to implement call-by-need evaluation. Split environments are a technique combining the two. To the best of our knowledge, these approaches are considered equivalently efficient. Here we show that in fact local environments admit a faster implementation, that can also be employed with split environments.

The complexity of abstract machines. There is a huge literature on abstract machines, but, apart from two key but isolated works by Blelloch and Greiner [17] and by Sands, Gustavsson, and Moran [38], complexity analyses of abstract machines have mostly been neglected. Motivated by recent advances on reasonable cost models for the λ-calculus by Accattoli and Dal Lago [10], in the last few years Accattoli and coauthors (Barenbaum, Guerrieri, Mazza, Sacerdoti Coen) [3–5, 7, 9, 11] have been developing a complexity-based theory of abstract machines, in which different techniques, approaches, and optimizations are classified depending on the complexity of their overhead. This paper belongs to this line of research, and at the same time it is the first step in a new direction.

Coq. The abstract machine at work in the kernel of Coq[1] [19] has been designed and partially studied by Barras in his PhD thesis [16], and provides a lightweight approach compared to the compilation scheme by Grégoire and Leroy described in [27]. It is used to decide the convertibility of terms, which is the bottleneck of the type-checking (and thus proof-checking) algorithm. It is at the same time one of the most sophisticated and one of the most used abstract machines for the λ-calculus. With this paper the authors initiate a research program aimed at developing the complexity analysis of the Coq main abstract machine. The goal is to prove it *reasonable*, that is, to show that the overhead of the machine is polynomial in the number of β-steps and in the size of the initial term, and eventually design a new machine along the way, if the existing one turns out to be unreasonable. Such a goal is challenging for various reason. For instance, the machine implements strong (*i.e.* under abstraction) call-by-need evaluation, whose formal operational semantics, due to Balabonski, Barenbaum, Bonelli, and Kesner, is finding its way into a published form just now [15]. Another reason is that it implements a language that is richer than the λ-calculus, see also the companion paper [6].

This paper. In this first step of our program, the aim is to provide a foundation for the unusual *split environments* at work in Barras' implementation. They were already used by Sestoft in [39] to describe a call-by-need abstract machine—our study can in fact be seen as a rational reconstruction and analysis of Sestoft's machine. The understanding of split environments for call-by-need is here built incrementally, by providing a fine analysis of the difference between local and global environments, of their implementations, and of their complexities. In particular, while the best implementation of call-by-need *requires* split environments, the properties of the different notions of environment do not depend on the evaluation strategy, and so we first present them in the simpler and more widely known setting of call-by-name evaluation. At the end of the paper, however, we apply the developed analysis to call-by-need evaluation, that is the case we are actually interested in. Our results also smoothly apply to call-by-value machines—we omit their study because it is modular and thus not particularly informative.

Contributions. The contributions of this paper are:

- *Implementation of global environments*: we provide the sketch of an OCaml implementation of global environments and of the Milner Abstract Machine (MAM, whose first appearance without a name is, we believe, in Sands, Gustavsson, and Moran's [38], and it has then been named MAM by Accattoli, Barenbaum, and Mazza in [4]), the simplest machine employing them. Not only we are not aware of any published implementation, but in our experience it is an implementation schema that is mostly unknown, even by experienced implementors. In particular, we implement the essential copy function in linear time.
- *Local environments are faster*: we analyze local environments in their most well-known incarnation, Krivine Abstract Machine (KAM), pointing out simple and yet unusual implementations, that—in the special case of terminating executions on closed terms—provide a better complexity than what is achievable with a global environment. Such an improvement is, to the best of our knowledge, new.
- *de Bruijn indices are slightly faster*: our fastest implementation scheme for local environments makes crucial use of de Bruijn indices. While the indices do not improve the overall complexity, they provide better bounds on some transitions. To our knowledge, this is the first theoretical evidence that de Bruijn indices provide a gain in efficiency.
- *Split environments*: we present a new machine with split environments, the *SPAM*, having the advantages of both the KAM and the MAM. In particular, the improved bound on local environments carries over to the split ones. We also provide an implementation in OCaml of the SPAM.
- *Call-by-need*: we recall a simple abstract machine for call-by-need from the literature, Accattoli, Barenbaum, and Mazza's *Pointing MAD* [4], and we reformulate it with split environments, obtaining the *Split MAD*, the backbone of the machine at work in Coq and very close to Sestoft's Mark 3 machine in [39]. For the Split MAD we show that our improved bound still holds, and we also provide an OCaml implementation.

Let us stress that the speed-up provided by local / split environments applies *only* to terminating weak evaluations of closed terms, that is the case of interest for functional programming languages such as Ocaml (call-by-value) or Haskell (call-by-need). The speed-up instead vanishes with open terms or strong evaluation (see the last paragraph of Sect. 9), that is, it does not apply to proof assistants, and in particular it does not apply to the Coq abstract machine.

The value of this paper. From a certain point of view the paper does not provide much original content. Essentially, most machines, analyses, and data structures at work in the paper already appeared in the literature. The value of the paper, then, is in connecting the dots, drawing a theory out of isolated results or techniques, putting them in perspective, providing a comprehensive study of environments, and surrounding it with a number of precious observations. The relevance of such an effort is witnessed by the fact that we obtain new bounds essentially for free.

For these reasons, and to stress the synthetic rather than the technical contribution of this work, most proofs are omitted (detailed proofs can be found in the literature, or can be obtained by minimal variations) and we provide OCaml code only for those cases that are not standard.

Related work. To our knowledge, there are no papers in the literature comparing the different styles of environments. The literature on abstract machines is however huge, let us just cite a few representative papers beyond those already mentioned [12, 13, 20, 21, 23, 25, 28, 31, 33, 37]. The literature on complexity analyses of abstract machines as already been cited. Call-by-need evaluation was introduced by Wadsworth [40] in the seventies. In the nineties, it was first reformulated as an operational semantics by Launchbury [32], Maraist, Odersky, and Wadler [34], and Ariola and Felleisen [14],

[1] The kernel of Coq is the subset of the codebase which ensures that only valid proofs are accepted. Hence the use of an abstract machine, which has a better ratio efficiency/complexity than the use of a compiler or a naive interpreter.

and then implemented by Sestoft [39]. For more recent work, see Chang and Felleisen's [18], Danvy and Zerny's [22], Garcia, Lumsdaine, and Sabry's [26], Pédrot and Saurin's [36], or the already cited work on strong call-by-need [15]. As already pointed out, our treatment of call-by-need is based on Accattoli, Barenbaum, and Mazza's [4], plus Accattoli and Sacerdoti Coen's [11].

2 PRELIMINARIES

λ-Calculus. The syntax of the ordinary λ-calculus is given by the following grammar for terms:

$$\lambda\text{-Terms} \qquad t, p, u, q ::= x \mid \lambda x.t \mid tp.$$

We use $t\{x{\leftarrow}p\}$ for the usual (meta-level) notion of substitution (of p for x in t). An abstraction $\lambda x.t$ binds x in t, and we silently work modulo α-renaming of bound variables, *e.g.* $(\lambda y.(xy))\{x{\leftarrow}y\} = \lambda z.(yz)$. We use $\mathtt{fv}(t)$ for the set of free variables of t. A term t is *closed* if it has no free variables (*i.e.* $\mathtt{fv}(t) = \emptyset$).

Call-by-name evaluation. The notion of evaluation we consider in this first part of the paper is the simplest one, that is, Plotkin's call-by-name strategy, also known as *weak head β-reduction.* Evaluation contexts are simply given by:

$$\text{CbN Evaluation Contexts} \qquad C ::= \langle\cdot\rangle \mid Ct$$

Then the strategy is defined by:

$$\begin{array}{cc} \text{Rule at top level} & \text{Contextual closure} \\ (\lambda x.t)p \mapsto_{\mathtt{CbN}} t\{x{\leftarrow}p\} & C\langle t\rangle \to_{\mathtt{CbN}} C\langle p\rangle \quad \text{if } t \mapsto_{\mathtt{CbN}} p \end{array}$$

A term t is a *normal form*, or simply *normal*, if there is no p such that $t \to_{\mathtt{CbN}} p$, and it is *neutral* if it is normal and it is not of the form $\lambda x.p$, *i.e.* it is not an abstraction. A *derivation* $d : t \to^k p$ is a finite, possibly empty, sequence of evaluation steps (also called reduction or rewriting steps). We write $|t|$ for the size of t and $|d|$ for the length of d.

Machines. We introduce general notions about abstract machines, given with respect to a generic machine $\mathtt{M}$ and a generic strategy $\to$ on λ-terms.

- An abstract machine $\mathtt{M}$ is given by *states*, noted s, and *transitions* between them, noted $\rightsquigarrow_{\mathtt{M}}$;
- A state is given by the *code under evaluation* plus some *data-structures*;
- The code under evaluation, as well as the other pieces of code scattered in the data-structures, are λ-terms *not considered modulo α-equivalence*;
- Codes are over-lined, to stress the different treatment of α-equivalence;
- A code $\overline{t}$ is *well-named* if x may occur only in $\overline{p}$ (if at all) for every sub-code $\lambda x.\overline{p}$ of $\overline{t}$;
- A state s is *initial* if its code is well-named and its data-structures are empty;
- Therefore, there is a bijection $\cdot^\circ$ (up to α) between terms and initial states, called *compilation*, sending a term t to the initial state t° on a well-named code α-equivalent to t;
- An *execution* is a (potentially empty) sequence of transitions $t_0^\circ \rightsquigarrow_{\mathtt{M}}^* s$ from an initial state obtained by compiling an (initial) term t_0;
- A state s is *reachable* if it can be obtained as the end state of an execution;
- A state s is *final* if it is reachable and no transitions apply to s;
- A machine comes with a map $\underline{\cdot}$ from states to terms, called *decoding*, that on initial states is the inverse (up to α) of compilation, *i.e.* $\underline{t^\circ} = t$ for any term t;
- A machine $\mathtt{M}$ has a set of *β-transitions*, whose union is noted $\rightsquigarrow_\beta$, that are meant to be mapped to β-redexes by the decoding, while the remaining *overhead transitions*, denoted by $\rightsquigarrow_o$, are mapped to equalities;
- We use $|\rho|$ for the length of an execution ρ, and $|\rho|_\beta$ for the number of β-transitions in ρ.

Implementations. Every abstract machine implements the strategy in the λ-calculus it was conceived for—this is usually expressed by an implementation theorem. Our notion of implementation, tuned towards complexity analyses, requires a perfect match between the number of β-steps of the strategy and the number of β-transitions of the machine.

Definition 2.1 (Machine implementation). A machine $\mathtt{M}$ *implements a strategy* $\to$ on λ-terms via a decoding $\underline{\cdot}$ when given a λ-term t the following holds:

(1) *Executions to derivations*: for any $\mathtt{M}$-execution $\rho\colon t^\circ \rightsquigarrow_{\mathtt{M}}^* s$ there exists a $\to$-derivation $d\colon t \to^* \underline{s}$.
(2) *Derivations to executions*: for every $\to$-derivation $d\colon t \to^* p$ there exists a $\mathtt{M}$-execution $\rho\colon t^\circ \rightsquigarrow_{\mathtt{M}}^* s$ such that $\underline{s} = p$.
(3) *β-Matching*: in both previous points the number $|\rho|_\beta$ of β-transitions in ρ is exactly the length $|d|$ of the derivation d, that is $|d| = |\rho|_\beta$.

Sufficient condition for implementations. The proofs of implementation theorems tend to follow always the same structure, based on a few abstract properties collected here into the notion of implementation system.

Definition 2.2 (Implementation system). A machine $\mathtt{M}$, a strategy $\to$, and a decoding $\underline{\cdot}$ form an *implementation system* if the following conditions hold:

(1) *β-Projection*: $s \rightsquigarrow_\beta s'$ implies $\underline{s} \to \underline{s'}$;
(2) *Overhead transparency*: $s \rightsquigarrow_o s'$ implies $\underline{s} = \underline{s'}$;
(3) *Overhead transitions terminate*: $\rightsquigarrow_o$ terminates;
(4) *Determinism*: both $\mathtt{M}$ and $\to$ are deterministic;
(5) *Progress*: $\mathtt{M}$ final states decode to $\to$-normal terms.

Theorem 2.3 (Sufficient condition for implementations, [9]). *Let* $(\mathtt{M}, \to, \underline{\cdot})$ *be an* implementation system. *Then,* $\mathtt{M}$ *implements* $\to$ *via* $\underline{\cdot}$.

3 GLOBAL CALL-BY-NAME: MILNER ABSTRACT MACHINE

Here we introduce the Milner Abstract Machine (MAM) [4, 38], a machine with a single *global* environment for call-by-name evaluation.

Machine components. The MAM is defined in Fig. 1. A machine state s is a triple $(\overline{t}, \pi, E)$ given by:

- *Code $\overline{t}$*: a term not considered up to α-equivalence, which is why it is over-lined;

$$\begin{array}{rrcl} \text{Environments} & E & := & \epsilon \mid [x{\leftarrow}\bar{t}] :: E \\ \text{Stacks} & \pi & := & \epsilon \mid \bar{t} :: \pi \\ \text{Compilation} & t^\circ & := & (\bar{t}, \epsilon, \epsilon) \end{array}
\quad\Bigg|\quad
\begin{array}{lrcl} \text{Decoding} & \underline{(\bar{t}, \epsilon, \epsilon)}_M & := & t \\ & \underline{(\bar{t}, \bar{p} :: \pi, E)}_M & := & \underline{(\bar{t}\bar{p}, \pi, E)}_M \\ & \underline{(\bar{t}, \epsilon, [x{\leftarrow}\bar{p}] :: E)}_M & := & \underline{(\bar{t}\{x{\leftarrow}\bar{p}\}, \epsilon, E)}_M \end{array}$$

Code	*Stack*	*Global Env*	*Trans*	*Code*	*Stack*	*Global Env*
$\bar{t}\bar{p}$	π	E	$\rightsquigarrow_{\mathsf{c}}$	$\bar{t}$	$\bar{p} :: \pi$	E
$\lambda x.\bar{t}$	$\bar{p} :: \pi$	E	$\rightsquigarrow_{\beta}$	$\bar{t}$	π	$[x{\leftarrow}\bar{p}] :: E$
x	π	$E :: [x{\leftarrow}\bar{t}] :: E'$	$\rightsquigarrow_{\mathsf{s}}$	$\bar{t}^{\alpha}$	π	$E :: [x{\leftarrow}\bar{t}] :: E'$

where $\bar{t}^{\alpha}$ denotes $\bar{t}$ where bound names have been freshly renamed.

Figure 1: Milner Abstract Machine (MAM).

- *Argument stack* π: it contains the arguments of the current code.
- *Global environment* E: a list of explicit (*i.e.* delayed) substitutions storing substitutions generated by the redexes encountered so far. It is used to implement micro-step substitution, *i.e.* substitution on one variable occurrence at a time.

Transitions. In the MAM there is one β-transition whereas overhead transitions are divided up into substitution and commutative transitions.

- *β-Transition* $\rightsquigarrow_{\beta}$: it morally fires a $\rightarrow_{\mathtt{CbN}}$-redex, the one corresponding to $(\lambda x.\bar{t})\bar{p}$, except that it puts a new delayed substitution $[x{\leftarrow}\bar{p}]$ in the environment instead of doing the meta-level substitution $\bar{t}\{x{\leftarrow}\bar{p}\}$ of the argument in the body of the abstraction;
- *Substitution transition* $\rightsquigarrow_{\mathsf{s}}$: it substitutes the variable occurrence under evaluation with a (properly α-renamed copy of a) code from the environment. It is a micro-step variant of meta-level substitution. It is invisible on the calculus because the decoding produces the term obtained by meta-level substitution, and so the micro work done by $\rightsquigarrow_{\mathsf{s}}$ cannot be observed at the coarser granularity of the calculus.
- *Commutative transition* $\rightsquigarrow_{\mathsf{c}}$: it locates and exposes the next redex according to the call-by-name strategy. It is invisible on the calculus.

Garbage collection: it is here simply ignored, or, more precisely, it is encapsulated at the meta-level, in the decoding function. It is well-known that this is harmless for the study of time complexity.

Compiling, decoding, and invariants. A term t is compiled to the machine *initial state* $t^\circ = (\bar{t}, \epsilon, \epsilon)$, where $\bar{t}$ is a well-named term α-equivalent to t. Conversely, every machine state s decodes to a term $\underline{s}_M$ (see the top right part of Fig. 1), obtained by first applying the code to the arguments in the stack π, and then applying the meta-level substitutions corresponding to the entries in the global environment E.

Implementation theorem. By means of omitted but essential invariants of the MAM (see [2] for details), one can prove that the hypotheses of Theorem 2.3 hold with respect to the call-by-name strategy, obtaining the following implementation theorem.

THEOREM 3.1 (MAM IMPLEMENTATION, [2, 4]). *The MAM implements call-by-name evaluation* $\rightarrow_{\mathtt{CbN}}$ *(via the decoding* $\underline{\cdot}_M$*).*

4 INTRODUCING COMPLEXITY ANALYSES

In this section we introduce the fundamental principles and a recipe for complexity analyses of abstract machines. As in Sect. 2, we refer to a generic machine $\mathtt{M}$ implementing a strategy $\rightarrow$ according to Def. 2.2, because in the next sections the recipe will be applied to various machines for various strategies.

Parameters for complexity analyses. By the *derivations-to-executions* part of the implementation (Point 2 in Def. 2.1), given a derivation $d\colon t_0 \rightarrow^n p$ there is a shortest execution $\rho\colon t_0^\circ \rightsquigarrow_{\mathtt{M}}^* s$ such that $\underline{s} = p$. Determining *the complexity of a machine* $\mathtt{M}$ amounts to bound the complexity of a concrete implementation of ρ on a RAM model, as a function of two fundamental parameters:

(1) *Input*: the size $|t_0|$ of the initial term t_0 of the derivation d;
(2) *β-Steps/transitions*: the length $n = |d|$ of the derivation d, that coincides with the number $|\rho|_\beta$ of β-transitions in ρ by the β-matching requirement for implementations (Point 3 in Def. 2.1).

A machine is *reasonable* if its complexity is polynomial in $|t_0|$ and $|\rho|_\beta$, and it is *efficient* if it is linear in both parameters.

Recipe for complexity analyses. For complexity analyses on a machine $\mathtt{M}$, overhead transitions $\rightsquigarrow_o$ are further separated into two classes, as it was the case for the MAM in the previous section:

(1) *Substitution transitions* $\rightsquigarrow_{\mathsf{s}}$: they are in charge of the substitution process;
(2) *Commutative transitions* $\rightsquigarrow_{\mathsf{c}}$: they are in charge of searching for the next β or substitution redex to reduce.

Then, the estimation of the complexity of a machine is done in three steps:

(1) *Number of transitions*: bounding the length of the execution ρ, by bounding the number of overhead transitions. This part splits into two subparts:
   (a) *Substitution vs β*: bounding the number $|\rho|_{\mathsf{s}}$ of substitution transitions in ρ using the number of β-transitions;
   (b) *Commutative vs substitution*: bounding the number $|\rho|_{\mathsf{c}}$ of commutative transitions in ρ using the size of the input and $|\rho|_{\mathsf{s}}$; the latter—by the previous point—induces a bound with respect to β-transitions.

(2) *Cost of single transitions*: bounding the cost of concretely implementing a single transition of $\mathtt{M}$. Here it is usually necessary to go beyond the abstract level, making some (high-level) assumption on how codes and data-structure are concretely represented. Commutative transitions are designed on purpose to have constant cost. Each substitution transition has a cost linear in the size of the initial term thanks to an invariant (to be proved) ensuring that only subterms of the initial term are duplicated and substituted along an execution. Each β-transition has a cost either constant or linear in the input.

(3) *Complexity of the overhead*: obtaining the total bound by composing the first two points, that is, by taking the number of each kind of transition times the cost of implementing it, and summing over all kinds of transitions.

(Linear) logical reading. Let us mention that our partitioning of transitions into β, substitution, and commutative ones admits a proof-theoretical view, as machine transitions can be seen as cut-elimination steps [4, 13]. Commutative transitions correspond to commutative cases, while β and substitution are principal cases. Moreover, in linear logic the β transition corresponds to the multiplicative case while the substitution transition to the exponential one. See [4] for more details.

5 COMPLEXITY OF THE MAM

The analysis of the MAM is well-known in the literature: it can be traced back to Sands, Gustavsson, and Moran's [38], it was then refined and decomposed in two parts by Accattoli and coauthors in [4, 8], and finally didactically treated by Accattoli in [2]. We present it here according to the recipe in three steps given in Sect. 4.

Step 1: number of transitions. Let $\rho : t_0^\circ \rightsquigarrow^* s$ be a MAM execution. We have (more details in [2]):

(1) *Substitution vs β*: $|\rho|_\mathtt{s} = O(|\rho|_\beta^2)$;
(2) *Commutative vs substitution (vs β)*: $|\rho|_\mathtt{c} = O(|\rho|_\mathtt{s} \cdot |t_0|)$, and so $|\rho|_\mathtt{c} = O(|\rho|_\beta^2 \cdot |t_0|)$;

The first bound is obtained via a standard reasoning building on the following easy facts:

(1) *Local bound*: the length of a maximal sequence of consecutive substitution transitions (eventually with commutatives in between, that however are not counted) is bound by the size $|E|$ of the global environment, because these substitution can only access E from left to right, for scoping reasons;
(2) *Environment size invariant*: the size $|E|$ of the global environment is bound by the number of preceding β-transitions, because they are the only ones to extend E.

The second bound is obtained by noting that:

(1) *Local bound*: the length of a maximal sequence of consecutive commutative transitions (eventually with β-transitions in between, that however are not counted) is bound by the size of the code, because commutatives decrease it;
(2) *Subterm invariant*: all pieces of code scattered in the data structures are subterms of the initial term t_0 (forthcoming Lemma 5.1).
(3) *Global bound*: the size of the code is increased only by substitution steps, but of at most $|t_0|$, by the subterm invariant.

Step 2: cost of single transitions. This is the part of the analysis on which we focus our attention in this paper. We have to give some details about the data structures for codes, stacks, and global environments. In this section we stay high-level, only describing the abstract properties of the data structures, while the next section discusses how to concretely realize them.

For stacks π, there is no special requirement: commutative and β transitions require to be able to do push and pop. A priori, there is no useful bound on the size of the stack. Therefore, it is natural to use a linked list implementation of stacks, for which push and pop are $O(1)$ operations, that in turn implies that $\rightsquigarrow_\mathtt{c}$ is $O(1)$.

For the global environment E, things are subtler. It is extended only on top by $\rightsquigarrow_\beta$, but it has to be accessed randomly (that is, not necessarily on top) by $\rightsquigarrow_\mathtt{s}$. There is a bound on its size (namely $|E| \leq |\rho|_\beta$, obtained simply observing that E is extended only by $\rightsquigarrow_\beta$), but going through E sequentially is expensive. Efficient implementations of E access its nth element directly, *i.e.* in constant time, by implementing E as a store, thus ignoring its list structure. In turn, this choice impacts on the data structure for codes, because it forces variables to be implemented as memory locations.

With these hypotheses $\rightsquigarrow_\beta$ can be implemented in $O(1)$. For $\rightsquigarrow_\mathtt{s}$, we need to take care of another important point, that is the implementation of α-renaming. There are two aspects: we need both a bound on the size of the code $\bar{t}$ to rename and a bound on the renaming operation $\bar{t}^\alpha$. The former is given by the following fundamental *subterm invariant*, a property of most implementations of the λ-calculus:

Lemma 5.1 (MAM subterm invariant). *Let ρ: $t_0^\circ \rightsquigarrow^* (\bar{t}, \pi, E)$ be a MAM execution. Then the size of every subcode $\bar{p}$ of $\bar{t}$, π, or E is bound by the size of t_0.*

For the bound on the renaming operation $\bar{t}^\alpha$, that essentially is a copy operation for which *renaming* means reallocation of all the memory locations used for variables, we assume here that it can be done in linear time (*i.e.* $O(|\bar{t}|)$, and so $O(|t_0|)$ by the subterm invariant). A linear algorithm is indeed possible, but it requires some care—this point is discussed in detail in the next section.

To sum up, we have the following quantitative assumptions:

- *Global environment*: implemented as a store, with extension and access in $O(1)$;
- *Renaming operation*: linear in the input, *i.e.* $O(|t_0|)$.

implying that $\rightsquigarrow_\mathtt{s}$ can be implemented in $O(|t_0|)$.

Step 3: complexity of the overhead. Composing the previous two points, it follows that in an execution ρ the cost of each group of transitions is (note that substitution and commutative transitions have the same bound, but it is obtained in different ways):

- β: $O(|\rho|_\beta)$;
- *Substitution*: $O(|\rho|_\beta^2 \cdot |t_0|)$;
- *Commutative*: $O(|\rho|_\beta^2 \cdot |t_0|)$;

Therefore, the MAM is reasonable and its complexity is $O(|\rho|_\beta^2 \cdot |t_0|)$.

6 IMPLEMENTING GLOBAL ENVIRONMENTS

We propose an Objective Caml implementation[2] of the MAM. As discussed in Section 5, the global environment is better implemented by a store, and variables are represented by pointers to cells of the store, to ensure that access is done in constant time.

The data stored in a variable cell needs to express that variables are either substituted (i.e. have an entry in the global environment), or bound by a λ-abstraction. Hence, in our implementation, variables carry an optional substitution. For reasons explained later, variable may also have a *copy* status.

The type of term is mutually recursively defined with the type of variables and that of substitutions:

```
type term =
    Var of var          (* Variable occurrences*)
  | App of term * term        (* Applications *)
  | Lam of var * term         (* Abstractions *)
and var = { name:string; mutable subs:subs }
and subs = NotSub | Subs of term | Copy of var
```

Variables are intended to be compared up to pointer equality. The carried name is actually just for printing convenience. Fresh names can be generated by a `gensym` function (of type `string->string`, the input string serves as a hint for the fresh name) each time a new variable is allocated.

```
let mkvar x = {name=gensym x; subs=NotSub}
```

To ensure the soundness of the term representation, the following invariant needs to be enforced:

- the variable attached to a λ-abstraction must be in the `NotSub` status.

The mutable part of the variable is used to perform substitutions in a term. There are two kinds of substitutions:

`Subs` are regular substitutions, generated by β-reduction;
`Copy` are substitutions used to perform the copy/renaming of a term.

In order to ensure that a side-effect on the variable carried by an abstraction only affects variables it binds, we need to establish another invariant, that applies only to the code component of the state:

- all variables carried by a binder should appear nowhere else in the state, but in a subterm of that binder.[3]

Implementation of the MAM. The state of the machine is just a pair of a code and a stack. The implicit global environment is made of all substituted variables reachable from the terms of the state. Unreachable variables can be garbage-collected by the runtime.

```
type state = term * term list

let rec mam (st:state) : state =
  match st with
  | App(u,v),           stk -> mam (u,v::stk)
  | Var{subs=Subs t0}, stk -> mam (copy t0,stk)
  | Lam(x, u), (v::stk) ->
     x.subs <- Subs v;
     mam (u,stk)
  | (Lam _, [] | Var _, _) -> st
```

The first case corresponds to the commutative transition, the second case to the substitution transition, the third case to the β-transition, and the last case to final states. The `copy` operation performs a renaming of bound variables, in order to maintain the invariant on the code component.

Let us remark that the above implementation (as well as all implementations in this paper) is tail-recursive, thanks to the use of a stack (the second component of the state). Since Objective Caml optimizes tail-recursion, the machine only consumes constant space on the process execution stack.

The complexity of each case is $O(1)$, but the exponentiation rule which requires a renaming of bound variables (`copy`). It remains to see how this operation can be implemented with linear complexity.

Renaming bound variables of a term. We give two implementations of the term copy operation. Both traverse the term, they differ on how they deal with variable occurrences.

The first one is purely functional, and uses a renaming map, noted `renMap`, carrying the renamings generated so far.

```
let rec rename renMap t =
  match t with
  | App(u,v) -> App(rename renMap u, rename renMap v)
  | Lam(x,u) ->
     let y = mkvar x.name in
     Lam(y,rename ((x,y)::renMap) u)
  | Var x ->
     (try Var (List.assq x renMap)
      with Not_found -> t)
let copy = rename []
```

Since the search in `r` is linear, and the number of entries may be linear in the size of the term, the global complexity of the renaming operation is quadratic.

For term representations that do not rely on pointer equality, or if we ensure that all variables carry distinct names (so `x==y` iff `x.name=y.name`), we may improve this complexity by using balanced trees for the renaming map. Access complexity is logarithmic, so the renaming complexity is $O(|t| \cdot \log |t|)$ where t is the term to copy (bound by the size of the initial term because of the subterm invariant of Lemma 5.1).

Improving the renaming function. An implementation with better complexity can be given, inspired by graph copy algorithms. It consists in using the substitution field of variables to perform the renaming. Bound variables (in the `NotSub` status) are temporarily put in the `Copy` status with a link to the new name for the variable. This step breaks the invariant that bound variable shall always be in the `NotSub` status, but it is restored by the end of the copy process.

This new status should not be conflated with `Subs` (created by β-reductions) as the term we rename may already contain substituted variables, and the renaming operation should not interfere with these substitutions.

[2] The implementations proposed in this article can be found in the git repository http://github.com/barras/abstract-machines.

[3] We may restrict this invariant to the binders not appearing in the right subterm of an application, as the call-by-name strategy does not perform reduction in those contexts.

$$
\begin{array}{rcll}
\text{Local Env.} & e & := & \epsilon \mid [x{\leftarrow}c] :: e \\
\text{Closures} & c & := & (\bar{t}, e) \\
\text{Stacks} & \pi & := & \epsilon \mid c :: \pi \\
\text{States} & s & := & (c, \pi) \\
\text{Compilation} & t^{\circ} & := & ((\bar{t}, \epsilon), \epsilon)
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Closure Decoding} & \underline{(\bar{t}, \epsilon)}_K & := & t \\
 & \underline{(\bar{t}, [x{\leftarrow}c] :: e)}_K & := & \underline{(\bar{t}\{x{\leftarrow}\underline{c}_K\}, e)}_K \\
\text{State Decoding} & \underline{(c, \epsilon)}_K & := & \underline{c}_K \\
 & \underline{(c, c' :: \pi)}_K & := & \underline{((\underline{c}_K \underline{c'}_K, \epsilon), \pi)}_K
\end{array}
$$

Code	*Local Env*	*Stack*	*Trans*	*Code*	*Local Env*	*Stack*
$\bar{t}\bar{p}$	e	π	$\rightsquigarrow_{\mathtt{c}}$	$\bar{t}$	e	$(\bar{p}, e) :: \pi$
$\lambda x.\bar{t}$	e	$c :: \pi$	$\rightsquigarrow_{\beta}$	$\bar{t}$	$[x{\leftarrow}c] :: e$	π
x	e	π	$\rightsquigarrow_{\mathtt{s}}$	$\bar{t}$	e'	π

with $e(x) = (\bar{t}, e')$

Figure 2: Krivine Abstract Machine (KAM).

```
let rec copy t =
  match t with
  | App(u,v) -> App(copy u, copy v)
  | Lam(x,u) ->
     let y = mkvar x.name in
     x.subs <- Copy y;
     let uWithXRenamedY = copy u in
     x.subs <- NotSub;
     Lam(y,uWithXRenamedY)
  | Var{subs=Copy y} -> Var y
  | Var _ -> t
```

Applications are copied by copying recursively their subterms.[4] In the case of λ-abstractions we first create a fresh variable y, which is then substituted for x. Then we copy recursively the body and get a term $uWithXRenamedY$. Once this is done, we need to restore the state of variable x, and we can return the new λ-abstraction. Variables in the copy status are simply replaced (without any further copy, unlike in the exponential rule). Other kinds of variable are not affected.

The complexity of this algorithm is linear. Therefore, we have given an implementation of the MAM with the complexity established in the previous section.

7 LOCAL CALL-BY-NAME: KRIVINE ABSTRACT MACHINE

Accounting for names. The analysis of the MAM requires a careful treatment of names through a dedicated invariant (here omitted, see [2]), but the process of α-renaming is kept at the meta-level and used as a black-box, on-the-fly operation (the *rename* / *copy* functions of the previous section).

The majority of the literature on abstract machines, instead, adopts another mechanism. The idea is to use different data structures, to circumvent α-renaming altogether—the machine never renames nor introduce new names (but it does extend the data-structures). To be precise, there are two levels (often confused):

(1) *Removal of on-the-fly α-renaming*: in these cases the machine works on terms with variable names but it is designed in order to implement evaluation without ever α-renaming. Technically, the global environment of the MAM is replaced by many local environments, each one for every piece of code in the machine. The machine becomes more complex, in particular the non-trivial concept of closure (to be introduced shortly) is necessary.

(2) *Removal of names*: terms are represented using de Bruijn indexes (or de Bruijn levels), removing the problem of α-renaming altogether but sacrificing the readability of the machine and reducing its abstract character. Usually this level is built on top of the previous one.

We are now going to introduce Krivine Abstract Machine (keeping names, so at the first level), that also implements the weak head strategy. Essentially, it is a version of the MAM without on-the-fly α-renaming. The complexity analysis is slightly different. To our knowledge, moreover, the complexities of the MAM and the KAM were considered to be the same, while Sect. 9 shows that the KAM can be implemented more efficiently.

Krivine abstract machine. The machine is in Fig. 2. It relies on the mutually inductively defined concepts of *local environment*, that is a list of closures, and *closure*, that is a pair of a code and a local environment. A state is a *pair* of a closure and a stack, but in the description of the transitions we write it as a *triple*, by spelling out the two components of the closure. Let us explain the name *closure*: usually, machines are executed on closed terms, and then a closure decodes indeed to a closed term. In the next section, we add the closed hypothesis to obtain our improved bounds, but for now it is superfluous (but *closures* keep their name, even without the closed hypothesis).

Garbage collection. Transition $\rightsquigarrow_{\mathtt{s}}$, beyond implementing micro-substitution, also accounts for some garbage collection, as it throws away the local environment e associated to the replaced variable x. The MAM simply ignores garbage collection. For time analyses garbage collection can indeed be safely ignored, while it is clearly essential for space. Both the KAM and the MAM are however desperately inefficient with respect to space.

Implementation. The proof that the KAM implements the weak head strategy is a classic result that can be proved by following the recipe for these proofs that we provided in Sect. 2, and it is omitted (see [2, 4], for instance).

[4] We may follow-up on the remark that the invariant only applies to binders on the left branch of the code component, by only renaming binders on that leftmost branch, although the others branches need to be copied in any case. We thereby avoid many useless renamings.

Theorem 7.1 (KAM implementation). *The KAM implements call-by-name evaluation $\rightarrow_{\mathtt{CbN}}$ (via the decoding $\underline{\cdot}_K$).*

8 COMPLEXITY OF THE KAM

Here we follow the recipe in three steps given in Sect. 4, but we somewhat do the second and the thirds steps at the same time.

Number of transitions. The bound of the number of transitions can be shown to be exactly as for the MAM. It is interesting to point out that the bound on $\rightsquigarrow_{\mathtt{s}}$ transitions requires a slightly different reasoning, because it is not possible to exploit the size of the global environment. It is enough to use a similar but slightly trickier invariant: the *depth* (*i.e.* the maximum nesting) of local environments is bound by the number of β-transitions. The proof is straightforward and omitted.

Cost of single transitions and complexity of the overhead. This is the interesting part of the analysis. It is based on the following invariant, that provides a bound on the length $|e|$ of local environments (defined as the number of items in e, seen as a list), and that is proved by a straightforward induction on executions.

Lemma 8.1 (Local environment size invariant). *Let $\rho : t_0^\circ \rightsquigarrow^* s$ be a KAM execution and $(\overline{p}, e)$ a closure in s. Then $|\overline{p}| + |e| \leq |t_0|$ (and so $|e| \leq |t_0|$).*

The bound can be slightly refined, because $|e|$ is actually bounded by the maximum number of abstractions on a branch of t_0 (seen as a tree)—in the worst case however this is no better than $|t_0|$.

From an abstract point of view, local environments are maps indexed by variables, with only two operations:

- *Push*: adding an item for a given variable, required by the β-transition;
- *Lookup*: retrieving the item associated to a given variable, required by the substitution transition.

Complexity analyses of abstract machines are a new topic, almost exclusively carried out by the first author and his coauthors. In their work they have generally assumed local environments to be implemented as lists, as it is the case in all implementations we are aware of. With these hypotheses, *push* is constant-time, and *lookup* takes $O(|t_0|)$ by Lemma 8.1, and the complexity of single transitions is exactly as for the MAM, giving also the same $O(|\rho|_\beta^2 \cdot |t_0|)$ complexity for the overhead of the KAM.

This paper stems from the observation that, despite common practice, local environments admit better implementations than lists. Now, we outline two implementations that lower the complexity of substitution transitions. In the general case, however, the overall complexity of the machine does not improve, because the cost of commutative transitions dominates over the substitution ones. The gain is nonetheless useful: in the specific case of terminating executions of closed terms—that is the relevant case for programming languages, actually—commutative transitions admit a better bound and the complexity of the KAM does improve.

Better implementation 1: balanced trees. First of all, local environments can be implemented in a purely functional way using balanced trees, obtaining *push* and *lookup* in $O(\log |t_0|)$. This choice improves the bound on the cost of substitution transitions, lowering it from $O(|\rho|_\beta^2 \cdot |t_0|)$ to $O(|\rho|_\beta^2 \cdot \log |t_0|)$, but it worsens the bound on β-transitions, raising it from $O(|\rho|_\beta)$ to $O(|\rho|_\beta \cdot \log |t_0|)$. The overall complexity stays the same, however, because the cost of commutative transitions does not change—they are still bound by $O(|\rho|_\beta^2 \cdot |t_0|)$—and dominates.

Better implementation 2: de Bruijn indices and random-access lists. Two ingredients are needed in order to improve further. First, environments can be implemented using Okasaki's *random-access lists* [35], that implement all list operations (push and pop) in constants time, and random-access (*i.e. lookup*) or update in logarithmic time ($O(\log |t_0|)$). This data structure represents lists as a list of perfect binary trees of growing size, and has very little overhead. Moreover, it is based on a beautifully simple idea—random-access lists are a *pearl* everyone should know about, we strongly suggest to read Okasaki's paper. Second, to take advantage of random-access lists, variable occurrences need to carry the index of their associated substitution in the local environment. The natural solution is to use de Bruijn indices to represent terms. In this way one obtains that the global cost of substitution transitions is $O(|\rho|_\beta^2 \cdot \log |t_0|)$ (as for balanced trees) and that of β-transitions is $O(|\rho|_\beta)$ (lower than balanced trees). Commutatives still dominate.

Worse implementation: arrays. For the sake of completeness, let us just mention that if local environments are implemented with arrays then *push* is $O(|t_0|)$ and *lookup* is constant time (if terms are represented using de Bruijn indices), but then the overhead becomes $O(|\rho|_\beta^2 \cdot |t_0|^2)$ (because a single commutative transition now costs $O(|t_0|)$), that is worse than for the MAM.

9 IMPROVING THE BOUND

Closed terms and terminating executions. Our recipe for complexity analyses of abstract machines is very general, it works for *every* execution, in particular for diverging executions and for any prefix of a terminating execution, and it does not make hypotheses on terms. The main case of interest in the study of programming languages, however, is the one of successful (*i.e.* terminating) executions for closed terms. With these hypotheses a better bound on the commutative transitions is possible: they depend only on, and actually coincide with, the number of β-transitions, that is $|\rho|_{\mathtt{c}} = |\rho|_\beta$—the dependency from the size of the initial term surprisingly disappears.

First of all, we have the following correlation between commutative and β-transitions, that is just the fact that commutatives push entries on the stack, while β-transitions pop them.

Proposition 9.1. *Let $\rho : s = \overline{t} \mid e \mid \pi \rightsquigarrow^* \overline{p} \mid e' \mid \pi' = s'$ be a KAM execution. Then $|\rho|_{\mathtt{c}} = |\rho|_\beta + |\pi'| - |\pi|$.*

Proof. By induction on ρ. If ρ is empty then $\pi = \pi'$ and the statement trivially holds. Otherwise ρ is $\tau : s \rightsquigarrow^* \overline{u} \mid e'' \mid \pi'' = s''$ followed by a transition $s'' \rightsquigarrow s'$. Cases of the last transition:

- *Commutative*: $|\rho|_{\mathtt{c}} = |\tau|_{\mathtt{c}} + 1 =_{i.h.} |\tau|_\beta + |\pi''| - |\pi| + 1 = |\rho|_\beta + |\pi''| - |\pi| + 1 = |\rho|_\beta + |\pi'| - |\pi|$.
- β: $|\rho|_{\mathtt{c}} = |\tau|_{\mathtt{c}} =_{i.h.} |\tau|_\beta + |\pi''| - |\pi| = |\rho|_\beta - 1 + |\pi''| - |\pi| = |\rho|_\beta - 1 + |\pi'| + 1 - |\pi| = |\rho|_\beta - 1 + |\pi'| + 1 - |\pi|$.
- *Substitution*: nothing changes. □

Note that Proposition 9.1 is not specific to the KAM, the same equality holds also for the MAM. Let us explain the connection between the bound of Proposition 9.1 and the $O(|\rho|_\beta^2 \cdot |t_0|)$ bound of Sect. 5: it is the size of π' that is bound by $O(|\rho|_\beta^2 \cdot |t_0|)$.

Now, it is well-known that if one considers a closed initial term then the normal form, when it exists, is an abstraction. Therefore, the final state of the machine has the form $\lambda x.\overline{p} \mid e \mid \epsilon$. Since initial states also have an empty stack, we obtain:

Corollary 9.2. *Let t_0 be a closed term and $\rho : t_0^\circ \rightsquigarrow^* s'$ be a KAM execution ending on a final state. Then $|\rho|_{\mathsf{c}} = |\rho|_\beta$.*

This fact is already used used by Sands, Gustavsson, and Moran in [38], even if less consciously, and it can even be traced back to the balanced traces of Sestoft [39], even if he did not make any quantitative analyses.

The improved bound. If we reconsider now the results of the previous section we obtain that, for terminating executions on closed terms, the complexity of the KAM is $O(|\rho|_\beta^2 \cdot \log|t_0|)$, if local environments are implemented with balanced trees or random-access lists, because the commutatives no longer dominate. Then

Theorem 9.3. *The complexity of terminating executions on closed terms on the KAM is bound by $O(|\rho|_\beta^2 \cdot \log|t_0|)$.*

Removing the quadratic dependency from $|\rho|_\beta$. It is natural to wonder if the quadratic dependency from $|\rho|_\beta$ is optimal. This point has been studied at length, and the answer is both *yes* and *no*. *Yes*, because there are families of terms reaching that bound for both the KAM and the MAM. *No*, because both machines can be optimized as to have a linear dependency from $|\rho|_\beta$. The quadratic overhead is due to growing chains of renamings of the form $[x_1{\leftarrow}x_2][x_2{\leftarrow}x_3]\cdots[x_{n-1}{\leftarrow}x_n]$ in the environment (in the case of the MAM, and similarly for the KAM). The easiest way to avoid these chains is by employing *compacting β-transitions*, that is, by replacing $\rightsquigarrow_\beta$ with the following two β-transitions:

$\lambda x.\overline{t}$	e	$(y, e') :: \pi$	$\rightsquigarrow_{\beta_1}$	$\overline{t}$	$[x{\leftarrow}e'(y)] :: e$	π
$\lambda x.\overline{t}$	e	$c :: \pi$	$\rightsquigarrow_{\beta_2}$	$\overline{t}$	$[x{\leftarrow}c] :: e$	π

where in $\rightsquigarrow_{\beta_2}$ it is assumed that c is not of the form (y, e'). The cost of $\rightsquigarrow_{\beta_1}$ is $O(\log|t_0|)$, because it both pushes and lookups environments. This optimization appears at least in Wand's [41] (section 2), Friedman et al.'s [25] (section 4), the second author's PhD dissertation [16] (section 3.3.3), and Sestoft's [39] (section 4) motivated as an optimization about *space*. In Sands, Gustavsson, and Moran's [38], however, it is shown for the first time to lower the overhead for *time* from quadratic to linear (on the MAM). This observation on the time complexity was also made in the second author's dissertation [16] on examples, but with no proof. Accattoli and Sacerdoti Coen's [11] provides a detailed study of this issue.

Let us call *Compacting KAM* the machine obtained from the KAM by replacing $\rightsquigarrow_\beta$ with the compacting β-transitions. Then,

Theorem 9.4. *The complexity of terminating executions on closed terms on the Compacting KAM is bound by $O(|\rho|_\beta \cdot \log|t_0|)$.*

Let us point out that the hypothesis on closed terms is essential. Morally, the dependency from the initial term disappears because normal forms, that are necessarily abstractions, can be recognized in constant time. With open terms (essential in the implementation of proof assistants) normal forms are not necessarily abstractions. Their size depends linearly on the size of the initial term and it is mandatory to explore a term to be sure that it is normal—therefore, the improvement showed here is not possible. Consider for instance the family of open terms defined by $x^1 := x$ and $x^{n+1} := x^n x$. The term x^n is normal but the (Compacting) KAM executes however n commutative transitions and no β-transitions.

10 SPLIT CALL-BY-NAME: THE SPAM

Local environments admit faster implementations than global environments. Global environments however allow forms of sharing that are not possible with local environments. The typical example is the memoization used in call-by-need evaluation, where evaluation enters the environment and the computed result has to affect all environments with that entry—this is not possible with local environments.

Is it possible to combine the best of both techniques? Yes, with what we like to call *split environments*. The idea, roughly, is to have:

- simpler local environments e, that only carry renamings of the form $[x_1{\leftarrow}a_1]\cdots[x_k{\leftarrow}a_k]$ where $a_1, \ldots, a_k$ are taken from a distinguished set of variables;
- and a global environment E of the form $[a_1{\leftarrow}c_1]\cdots[a_k{\leftarrow}c_k]$, where a closure c has the form $(\overline{t}, e)$.

The *SPlit environments Abstract Machine* (*SPAM*), a new call-by-name machine in between the MAM and the KAM, is defined in Fig. 3. Note that $\rightsquigarrow_\beta$ requires a to be a fresh name. The fact that $a_1, \ldots, a_k$ are a distinguished set of variables is not required, but it is easily seen that this is an independent space of names—they can be thought as pointers to the global environment / store.

There is an easy decoding $\underline{\cdot}_S$ of SPAM states to MAM states (in Fig. 3), obtained by turning all closures into codes by applying local environments as substitutions. Such a decoding induces a strong bisimulation between the two machines (whose proof is straightforward), from which the implementation theorem for the SPAM immediately follows.

Proposition 10.1.

1. SPAM / MAM Strong Bisimulation*: let s be a SPAM reachable state and $l \in \{\mathsf{c}, \beta, \mathsf{s}\}$. Then $\underline{s}_S$ is a MAM reachable state and*
   - (a) *$s \rightsquigarrow_l s'$ implies $\underline{s}_S \rightsquigarrow_l \underline{s'}_S$;*
   - (b) *if $\underline{s}_S \rightsquigarrow_l s'$ then there is s'' such that $s \rightsquigarrow_l s''$ and $\underline{s''} = s'$.*
2. Implementation*: the SPAM implements call-by-name evaluation $\rightarrow_{\mathsf{CbN}}$ (via the decoding $\underline{\underline{\cdot}_S}_M$).*

Clearly, the same properties could be obtained by decoding SPAM states to KAM states, by substituting the global environment on the local ones.

SPAM implementation. The idea is that the local environment is implemented as a balanced tree or a random-access list, and the global environment as a store. We assume that generating a fresh name, that in fact is a fresh store location, is constant-time. On the one hand, this is reasonable because so it is on the RAM model. On the other hand, this is also implicit in the MAM (in the renaming operation) and in the KAM (the push on e creates a new entry in the data structure for e), and so it is not a feature of the SPAM.

$$\begin{array}{rccl}
\text{Closures} & e & := & (\bar{t}, e) \\
\text{Local Env.} & e & := & \epsilon \mid [x{\leftarrow}a] :: e \\
\text{Stacks} & \pi & := & \epsilon \mid c :: \pi \\
\text{Global Env.} & E & := & \epsilon \mid [a{\leftarrow}c] :: E \\
\text{States} & s & := & (c, \pi, E) \\
\text{Compilation} & t^{\circ} & := & ((\bar{t}, \epsilon), \epsilon, \epsilon)
\end{array}
\quad\Bigg|\quad
\begin{array}{rrcl}
\text{Closure Decoding} & \underline{(\bar{t}, \epsilon)}_S & := & \bar{t} \\
 & \underline{(\bar{t}, [x{\leftarrow}a] :: e)}_S & := & \underline{(\bar{t}\{x{\leftarrow}a\}, e)}_S \\
\text{Stack / Gl Env Decoding} & \underline{\epsilon}_S & := & \epsilon \\
 & \underline{c :: \pi}_S & := & \underline{c}_S :: \underline{\pi}_S \\
 & \underline{[a{\leftarrow}c] :: E}_S & := & [a{\leftarrow}\underline{c}_S] :: \underline{E}_S \\
\text{State Decoding} & \underline{(c, \pi, E)}_S & := & (\underline{c}_S, \underline{\pi}_S, \underline{E}_S)
\end{array}$$

Code	*Local Env*	*Stack*	*Global Env*	*Trans*	*Code*	*Local Env*	*Stack*	*Global Env*	
$\bar{t}\bar{p}$	e	π	E	$\rightsquigarrow_{\mathsf{c}}$	$\bar{t}$	e	$(\bar{p}, e) :: \pi$	E	
$\lambda x.\bar{t}$	e	$c :: \pi$	E	$\rightsquigarrow_{\beta}$	$\bar{t}$	$[x{\leftarrow}a] :: e$	π	$[a{\leftarrow}c] :: E$	with a fresh
x	e	π	E	$\rightsquigarrow_{\mathsf{s}}$	$\bar{t}$	e'	π	E	with $E(e(x)) = (\bar{t}, e')$

Figure 3: SPlit Abstract Machine (SPAM).

$$\begin{array}{rccl}
\text{Stacks} & \pi & := & \epsilon \mid \bar{t} :: \pi \\
\text{Dumps} & E & := & \epsilon \mid (x, \pi) :: D \\
\text{Global Env.} & E & := & \epsilon \mid [x{\leftarrow}\bar{t}] :: E \mid [x{\leftarrow}\square] :: E
\end{array}
\quad\Bigg|\quad
\begin{array}{rccl}
\text{States} & s & := & (\bar{t}, \pi, D, E) \\
\text{Compilation} & t^{\circ} & := & (\bar{t}, \epsilon, \epsilon, \epsilon)
\end{array}$$

Code	*Stack*	*Dump*	*Global Env*	*Trans*	*Code*	*Stack*	*Dump*	*Globla Env*
$\bar{t}\bar{p}$	π	D	E	$\rightsquigarrow_{\mathsf{c}_1}$	$\bar{t}$	$\bar{p} :: \pi$	D	E
$\lambda x.\bar{t}$	$\bar{p} :: \pi$	D	E	$\rightsquigarrow_{\beta}$	$\bar{t}$	π	D	$[x{\leftarrow}\bar{p}] :: E$
x	π	D	$E :: [x{\leftarrow}\bar{t}] :: E'$	$\rightsquigarrow_{\mathsf{c}_2}$	$\bar{t}$	ϵ	$(x, \pi) :: D$	$E :: [x{\leftarrow}\square] :: E'$
$\bar{v}$	ϵ	$(x, \pi) :: D$	$E :: [x{\leftarrow}\square] :: E'$	$\rightsquigarrow_{\mathsf{s}}$	$\bar{v}^{\alpha}$	π	D	$E :: [x{\leftarrow}\bar{v}] :: E'$

Figure 4: Pointing Milner Abstract machine by-neeD (Pointing MAD).

The bisimulation property implies that the SPAM and the MAM have the same number of transitions. Therefore, the complexity of the SPAM on closed terms and terminating executions is $O(|\rho|_{\beta}^{2} \cdot \log |t_0|)$. Exactly as for the KAM, one can define a Compacting SPAM of complexity $O(|\rho|_{\beta} \cdot \log |t_0|)$. Actually, with a global environment there is a lazier way of removing renamings chains (see Accattoli and Sacerdoti Coen's [7]) giving the *Unchaining SPAM*, that does not raise the complexity of β-transitions (but the overall complexity does not change).

We remark that, as for local environments, split environments are less sensitive to the representation of terms, yet allowing sharing: for the MAM we had to devise a clever term representation (involving subtle invariants about pointer sharing), to efficiently perform substitution. Machines with split environments can have a good complexity with many term representations: de Bruijn indices, name-carrying, etc. There is a slight gain in using de Bruijn indices, as we pointed out in Sect. 8, but the asymptotic complexity is the same.

11 IMPLEMENTING SPLIT ENVIRONMENTS

Split environment machines feature both a global and a local environment, the latter being a mapping from variables to pointers. As before, global environments are implemented by a store. We assume we have a module Env implementing local environments efficiently. The signature LocalEnv of the expected operations is:

```
module type LocalEnv =
sig
  type 'a t
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  val access : int -> 'a t -> 'a
end
```

where the type 'a t represents the type of local environments associating data of type 'a to each variable. Hence the type 'a Env.t is the type of local environments, and Env.empty, Env.push and Env.access are the operations associated to this type. We use the well-known type of λ-terms with de Bruijn indices. λ-abstractions carry a name only for printing convenience. Other term representations may be used with an impact on the efficiency of the machine, as already discussed, but the global complexity remains the same.

```
type term =
    Var of int  (* de Bruijn indices *)
  | App of term * term
  | Lam of string * term
```

The store contains mutable cells holding an optional *value*. A cell contains the special value Box when the reference is being evaluated. This will be used for the Call-by-Need strategy in Sect. 14.

```
type env = ptr Env.t
and ptr = value ref
and value = Box | Clos of (term * env)
```

The call-by-name SPAM can be implemented straightforwardly, following the rules of the KAM, see Fig. 2:

```
type stack = (term * env) list
type state = term * env * stack

let rec spam (st:state) : value =
  match st with
  | App(u,v), e, stk -> spam(u,e,(v,e)::stk)
  | Lam(_, u), e, (v::stk) ->
     spam(u, Env.push (ref (Clos v)) e, stk)
  | Var n, e, stk ->
     (match !(Env.access n e) with
     | Clos(t,e') -> spam(t,e',stk)
     | Box -> assert false)
  | (Lam _ as t, e, []) -> Clos(t,e)
```

The only difference with the KAM is the use of mutable references to encode the global environment. But the mutability is not exploited by the call-by-name strategy.

12 GLOBAL CALL-BY-NEED: THE POINTING MAD

Call-by-need evaluation is an enhancement of call-by-name evaluation with both a flavor of call-by-value and a form of sharing sometimes called memoization. The idea is that the first time that a piece of code in the environment is needed (because it has to replace a variable occurrence) it is evaluated before being used (call-by-value flavor), and moreover the value is stored to avoid its re-computation when it will be needed again (memoization).

The *Pointing Milner Abstract machine by-neeD* (*Pointing MAD*), a simple call-by-need machine with a global environment, is defined in Fig. 4, and it is taken from [4], where the interested reader can find an implementation theorem with respect to a call-by-need calculus. Memoization is naturally realized using a global environment, by updating its entries. The call-by-value flavor is instead realized by adding a further data structure, the *dump*, storing the sequence of environment entries traversed by evaluation to arrive at the code under evaluation, and by introducing the syntactic category of value, that are simply defined as abstractions. Let us explain the transitions of the Pointing MAD (except for $\rightsquigarrow_{\mathsf{c}_1}$, that is simply transition $\rightsquigarrow_{\mathsf{c}}$ of the MAM):

- $\rightsquigarrow_{\mathsf{c}_2}$: when the code $\overline{t}$ substituting a variable x is needed, the machine jumps into the entry $[x{\leftarrow}\overline{t}]$ associated to x in the environment and starts to evaluate $\overline{t}$. It also saves on the dump the current stack π and the variable x in which it entered, to backtrack when the evaluation of $\overline{t}$ will be over. Last, it marks the entry $[x{\leftarrow}\overline{t}]$ with a special symbol $\square$, meaning that the entry is being evaluated.
- $\rightsquigarrow_{\mathsf{s}}$: if evaluation entered in a substitution (and so the dump is non-empty) and evaluation is over (an abstraction $\overline{v}$ has been obtained, and the stack is empty) then it is time to backtrack out of the substitution. Concretely, the machine removes the first entry (x, π) from the dump, and it uses it to restore the old stack π and update the suspended substitution $[x{\leftarrow}\square]$ to $[x{\leftarrow}\overline{v}]$. Crucially, the transition also replaces the occurrence of x that made the machine jump inside the substitution in the first place with a renamed copy $\overline{v}^{\alpha}$ of the obtained value $\overline{v}$—this is where the substitution process takes place.
- $\rightsquigarrow_{\beta}$: it is the usual β-transition. Note that the Pointing MAD in [4] has two variants of $\rightsquigarrow_{\beta}$, but only to ease the correspondence with the calculus and the proof of the implementation theorem. The variants differ in how they treat the list structure of the global environment E. Since such a structure disappears in the implementation—because E is implemented as a store—here we use a simpler machine with just one β-transition.

Complexity analysis. Accattoli and coauthors have provided a detailed complexity analysis of the Pointing MAD. Let $\rho : t_0^{\circ} \rightsquigarrow^{*} s$ be a Pointing MAD execution. Then

- *Substitution vs β-transitions* [11]: $|\rho|_{\mathsf{s}} = O(|\rho|_{\beta})$;
- *Commutative vs substitution (vs β)* [4]: $|\rho|_{\mathsf{c}_2} = O(|\rho|_{\mathsf{s}} + |\rho|_{\beta})$, and so $|\rho|_{\mathsf{c}_2} = O(|\rho|_{\beta})$ while $|\rho|_{\mathsf{c}_1} = O(|\rho|_{\mathsf{s}} \cdot |t_0|)$, and so $|\rho|_{\mathsf{c}_1} = O(|\rho|_{\beta} \cdot |t_0|)$ as for the MAM / KAM. Then in general $|\rho|_{\mathsf{c}} = O(|\rho|_{\beta} \cdot |t_0|)$.

The same reasoning for terminating executions on closed terms done in Sect. 9 applies here to $\rightsquigarrow_{\mathsf{c}_1}$, and so in such a case we obtain $|\rho|_{\mathsf{c}} = O(|\rho|_{\beta})$, that implies $|\rho| = O(|\rho|_{\beta})$.

The Pointing MAD can be implemented like the MAM using the approach outlined in Sect. 6. A subterm invariant holds, and so the cost of a single substitution transition is $O(|t_0|)$, because the substituted value has to be renamed. All other transitions can be implemented in constant time. Then the complexity of the Pointing MAD is $O(|\rho|_{\beta} \cdot |t_0|)$, also if terminating executions and closed terms are considered, because in that case commutative transitions become faster but here substitution transitions still dominate.

13 SPLIT CALL-BY-NEED: THE SPLIT MAD

The next and final step is to apply the technology of split environments developed in Sect. 10 to the Pointing MAD of the previous section, obtaining the *Split MAD* in Fig. 5. Analogously to how the SPAM decodes to the MAM, the Split MAD decodes to the Pointing MAD via the decoding function $\underline{\cdot}_N$ in Fig. 5, and such a decoding induces a strong bisimulation, whose proof is straightforward.

Proposition 13.1 (Split MAD / Pointing MAD strong bisimulation). *Let s be Split MAD reachable state and $l \in \{\mathsf{c}_1, \beta, \mathsf{c}_2, \mathsf{s}\}$. Then $\underline{s}_N$ is a Pointing MAD reachable state and*

(1) $s \rightsquigarrow_l s'$ implies $\underline{s}_N \rightsquigarrow_l \underline{s'}_N$;
(2) if $\underline{s}_N \rightsquigarrow_l s'$ then there is s'' such that $s \rightsquigarrow_l s''$ and $\underline{s''} = s'$.

From the bisimulation immediately follows that the Split MAD implements call-by-need evaluation, but, since for the sake of conciseness we did not introduce a call-by-need calculus, we do not state such a theorem.

Complexity analysis. From the strong bisimulation property it follows that in terms of number of transitions the Split MAD behaves exactly as the Pointing MAD, *i.e.* $|\rho| = O(|\rho|_{\beta} \cdot |t_0|)$ in the general case (where $|t_0|$ is due to $\rightsquigarrow_{\mathsf{c}_1}$) and $|\rho| = O(|\rho|_{\beta})$ for terminating executions on closed terms.

Single $\rightsquigarrow_{\mathsf{c}_1}$ and $\rightsquigarrow_{\beta}$ transitions are constant-time, while $\rightsquigarrow_{\mathsf{s}}$ transitions are $O(\log |t_0|)$ because of split environments, that is an improvement over the Pointing MAD. Note that instead $\rightsquigarrow_{\mathsf{c}_2}$ now

$$\begin{array}{rcll}
\text{Closures} & e & := & (\bar{t}, e) \\
\text{Local Env.} & e & := & \epsilon \mid [x{\leftarrow}a] :: e \\
\text{Stacks} & \pi & := & \epsilon \mid c :: \pi \\
\text{Dumps} & E & := & \epsilon \mid (a, \pi) :: D \\
\text{Global Env.} & E & := & \epsilon \mid [a{\leftarrow}c] :: E \mid [a{\leftarrow}\square] :: E \\
\text{States} & s & := & (c, \pi, D, E) \\
\text{Compilation} & t^{\circ} & := & ((\bar{t}, \epsilon), \epsilon, \epsilon, \epsilon)
\end{array}$$

$$\begin{array}{rrcl}
\text{Closure Decoding} & \underline{(\bar{t}, \epsilon)}_N & := & \bar{t} \\
 & \underline{(\bar{t}, [x{\leftarrow}a] :: e)}_N & := & \underline{(\bar{t}\{x{\leftarrow}a\}, e)}_N \\
\text{Stack / Gl Env / Dump Decoding} & \underline{\epsilon}_N & := & \epsilon \\
 & \underline{c :: \pi}_N & := & \underline{c}_N :: \underline{\pi}_N \\
 & \underline{[a{\leftarrow}c] :: E}_N & := & [a{\leftarrow}\underline{c}_N] :: \underline{E}_N \\
 & \underline{(a, \pi) :: D}_N & := & (a, \underline{\pi}_N) :: \underline{D}_N \\
\text{State Decoding} & \underline{(c, \pi, D, E)}_N & := & (\underline{c}_N, \underline{\pi}_N, \underline{D}_N, \underline{E}_N)
\end{array}$$

Code	*LocEnv*	*Stack*	*Dump*	*Global Env*	*Trans*	*Code*	*LocEnv*	*Stack*	*Dump*	*Global Env*
$\bar{t}\bar{p}$	e	π	D	E	$\rightsquigarrow_{\mathsf{c}_1}$	$\bar{t}$	e	$(\bar{p}, e) :: \pi$	D	E
$\lambda x.\bar{t}$	e	$c :: \pi$	D	E	$\rightsquigarrow_{\beta}$	$\bar{t}$	$[x{\leftarrow}a] :: e$	π	D	$[a{\leftarrow}c] :: E$
x	e	π	D	$E :: [e(x){\leftarrow}(\bar{t}, e'')] :: E'$	$\rightsquigarrow_{\mathsf{c}_2}$	$\bar{t}$	e''	ϵ	$(e(x), \pi) :: D$	$E :: [e(x){\leftarrow}\square] :: E'$
$\bar{v}$	e	ϵ	$(a, \pi) :: D$	$E :: [a{\leftarrow}\square] :: E'$	$\rightsquigarrow_{\mathsf{s}}$	$\bar{v}$	e	π	D	$E :: [a{\leftarrow}(\bar{v}, e)] :: E'$

Figure 5: Split Milner Abstract machine by-neeD (Split MAD).

costs more than for the Pointing MAD, because to jump in the global environment (that before was constant-time) now the machine has to first access the local environment, that requires $O(\log |t_0|)$. Such a slowdown however does not impact on the overall complexity. To sum up, our main result is that

Theorem 13.2. *The complexity of terminating executions on closed terms on the Split MAD is bound by* $O(|\rho|_\beta \cdot \log |t_0|)$.

Sestoft’s Mark 3 machine in [39] is essentially the Split MAD plus the compacting β-transitions at the end of Sect. 9 (to be precise: Sestoft has only $\rightsquigarrow_{\beta_2}$ because he preprocesses terms in a administrative normal form to which only $\rightsquigarrow_{\beta_2}$ applies). Here the compacting transitions have no impact on the complexity, that is already linear in $|\rho|_\beta$, which is why we did not include them in the Split MAD. They do, however, make the environment more compact.

14 IMPLEMENTING THE SPLIT MAD

A naive way of implementing a call-by-need (or lazy) machine is to patch a call-by-name machine: it suffices, in the variable case, to first launch the machine on the computation associated to the variable and, upon return, update the value of the variable. However, this implementation is not tail-recursive and the interpreter may consume a lot of space on the process stack.

The implementation can be kept tail-recursive by adding a new component to the state (the dump). The dump collects pointers to shared computations that are being performed, and also the stack in which the result of the computation is to be placed. To reduce the space requirements, these pointers are set to a special value Box while the referenced computation is performed. This allows for early garbage-collection of unused temporary steps of the computation.

The Split MAD can be implemented straightforwardly, following the definition of Fig. 4. We reuse the definitions of terms, local environments and stacks of Sect. 11.

```
1 type dump = (ptr * stack) list
2 type dstate = term * env * stack * dump
3
4 let rec smad (st:dstate) : value =
5   match st with
6   | App(u,v), e, stk, d -> smad(u,e,(v,e)::stk,d)
7   | Lam(_,u), e, (v::stk), d ->
8      smad(u,Env.push(ref(Clos v)) e,stk,d)
9   | Var n,     e, stk, d ->
10      let p = Env.access n e in
11      (match !p with
12      | Clos(t,e') -> p:=Box; smad(t,e',[],(p,stk)::d)
13      | Box -> assert false)
14  | Lam _ as t, e, [], (p,stk)::d ->
15      p := Clos(t,e); smad(t,e,stk,d)
16  | (Lam _ as t, e, [], []) -> Clos(t,e)
```

Each recursive call correspond to one of the rule of the Split MAD: in order $\rightsquigarrow_{\mathsf{c}_1}$, $\rightsquigarrow_{\beta}$, $\rightsquigarrow_{\mathsf{c}_2}$ and $\rightsquigarrow_{\mathsf{s}}$. The last case correspond to the final state of the machine. All steps are in constant time but $\rightsquigarrow_{\mathsf{c}_2}$ which is logarithmic in the size of the initial term.

In a practical implementation, it would make sense to detect (in line 12) when the global environment yields a λ-abstraction, and avoid multiple useless side-effects and dump operations. Another valuable improvement would be to avoid (in line 6) closures which code is a variable and eagerly perform the local environment access, as in Sestoft [39]. This would require to change the type of stacks (now a list of pointers) and move the store allocation from the β-transition to the commutative one.

15 CONCLUSIONS

The paper provides a thorough study of local, global, and split environments, from both complexity and implementative point of views. In particular, it shows how the local ones admit a faster implementation in the special case of terminating executions on closed terms, which is the main case of interest for functional languages. Moreover, such an improvement carries over to split environments, and it is thus applicable to call-by-need evaluation, the strategy on which the Coq abstract machine is actually based.

Companion paper and future work. The next step in our program is to extend the complexity analysis to a richer call-by-need λ-calculus, closer to the one actually used by Coq abstract machine. The companion paper [6] studies the extension with pattern matching—which is similar to the algebraic datatype extensions studied by Sestoft in [39]—showing that its complexity analysis is trickier than expected. In future work, we will address fixpoints, open terms, and strong evaluation.

Acknowledgements. This work has been partially funded by the ANR JCJC grant COCA HOLA (ANR-16-CE40-004-01).

REFERENCES

[1] Beniamino Accattoli. 2016. COCA HOLA. https://sites.google.com/site/beniaminoaccattoli/coca-hola. (2016).

[2] Beniamino Accattoli. 2016. The Complexity of Abstract Machines. In *WPTE@FSCD 2016*. 1–15.

[3] Beniamino Accattoli. 2016. The Useful MAM, a Reasonable Implementation of the Strong λ-Calculus. In *WoLLIC 2016*. Springer, 1–21.

[4] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2014. Distilling Abstract Machines. In *ICFP 2014*. ACM, 363–376.

[5] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2015. A Strong Distillery. In *APLAS 2015*. Springer, 231–250.

[6] Beniamino Accattoli and Bruno Barras. 2017. The Negligible and Yet Subtle Cost of Pattern Matching. Accepted to APLAS 2017. (2017).

[7] Beniamino Accattoli and Claudio Sacerdoti Coen. 2015. On the Relative Usefulness of Fireballs. In *LICS 2015*. IEEE Computer Society, 141–155.

[8] Beniamino Accattoli and Ugo Dal Lago. 2012. On the Invariance of the Unitary Cost Model for Head Reduction. In *RTA 2012*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 22–37.

[9] Beniamino Accattoli and Giulio Guerrieri. 2017. Implementing Open Call-by-Value. Accepted at FSEN 2017. (2017).

[10] Beniamino Accattoli and Ugo Dal Lago. 2016. (Leftmost-Outermost) Beta Reduction is Invariant, Indeed. *Logical Methods in Computer Science* 12, 1 (2016). https://doi.org/10.2168/LMCS-12(1:4)2016

[11] Beniamino Accattoli and Claudio Sacerdoti Coen. 2014. On the Value of Variables. In *WoLLIC 2014*. Springer, 36–50.

[12] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2004. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inf. Process. Lett.* 90, 5 (2004), 223–232.

[13] Zena M. Ariola, Aaron Bohannon, and Amr Sabry. 2009. Sequent calculi and abstract machines. *ACM Trans. Program. Lang. Syst.* 31, 4 (2009), 13:1–13:48. https://doi.org/10.1145/1516507.1516508

[14] Zena M. Ariola and Matthias Felleisen. 1997. The Call-By-Need lambda Calculus. *J. Funct. Program.* 7, 3 (1997), 265–301.

[15] Thibaut Balabonski, Pablo Barenbaum, Eduardo Bonelli, and Delia Kesner. 2017. Foundations of Strong Call by Need. Accepted at ICFP 2017. (2017).

[16] Bruno Barras. 1999. *Auto-validation d'un système de preuves avec familles inductives.* Ph.D. Dissertation. Université Paris 7.

[17] Guy E. Blelloch and John Greiner. 1995. Parallelism in Sequential Functional Languages. In *FPCA 1995*. ACM, 226–237.

[18] Stephen Chang and Matthias Felleisen. 2012. The Call-by-Need Lambda Calculus, Revisited. In *ESOP 2012*. Springer, 128–147.

[19] Coq Development Team. 2016. The Coq Proof-Assistant Reference Manual, version 8.6. (2016). http://coq.inria.fr

[20] Pierre Crégut. 2007. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation* 20, 3 (2007), 209–230.

[21] Olivier Danvy and Lasse R. Nielsen. 2004. *Refocusing in Reduction Semantics*. Technical Report RS-04-26. BRICS.

[22] Olivier Danvy and Ian Zerny. 2013. A synthetic operational account of call-by-need evaluation. In *PPDP 2013*. ACM, 97–108.

[23] Matthias Felleisen and Daniel P. Friedman. 1986. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*.

[24] Maribel Fernández and Nikolaos Siafakas. 2009. New Developments in Environment Machines. *Electr. Notes Theor. Comput. Sci.* 237 (2009), 57–73.

[25] Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek, and Onnie Lynn Winebarger. 2007. Improving the lazy Krivine machine. *Higher-Order and Symbolic Computation* 20, 3 (2007), 271–293.

[26] Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. 2009. Lazy evaluation and delimited control. In *POPL 2009*. ACM, 153–164.

[27] Benjamin Grégoire and Xavier Leroy. 2002. A compiled implementation of strong reduction. In *ICFP 2002)*. ACM, 235–246.

[28] Thérèse Hardin and Luc Maranget. 1998. Functional Runtime Systems Within the Lambda-Sigma Calculus. *J. Funct. Program.* 8, 2 (1998), 131–176.

[29] Jean-Baptiste Jeannin and Dexter Kozen. 2012. Computing with Capsules. *Journal of Automata, Languages and Combinatorics* 17, 2-4 (2012), 185–204.

[30] Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation* 20, 3 (2007), 199–207.

[31] Peter John Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (Jan. 1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308

[32] John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *POPL 1993*. ACM Press, 144–154.

[33] Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language.* Technical report 117. INRIA. http://gallium.inria.fr/~xleroy/publi/ZINC.pdf

[34] John Maraist, Martin Odersky, and Philip Wadler. 1998. The Call-by-Need Lambda Calculus. *J. Funct. Program.* 8, 3 (1998), 275–317.

[35] Chris Okasaki. 1995. Purely Functional Random-Access Lists. In *FPCA 1995*. ACM, 86–95.

[36] Pierre-Marie Pédrot and Alexis Saurin. 2016. Classical By-Need. In *ESOP 2016*. Springer, 616–643. https://doi.org/10.1007/978-3-662-49498-1_24

[37] Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159.

[38] David Sands, Jörgen Gustavsson, and Andrew Moran. 2002. Lambda Calculi and Linear Speedups. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*. Springer, 60–84.

[39] Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. *J. Funct. Program.* 7, 3 (1997), 231–264.

[40] Christopher P. Wadsworth. 1971. *Semantics and pragmatics of the lambda-calculus.* PhD Thesis. Oxford. Chapter 4.

[41] Mitchell Wand. 2007. On the correctness of the Krivine machine. *Higher-Order and Symbolic Computation* 20, 3 (2007), 231–235.