Positive Sharing and Abstract Machines

Beniamino Accattoli[1], Claudio Sacerdoti Coen[2], and Jui-Hsuan Wu[3]

[1] Inria & LIX, École Polytechnique, UMR 7161, Palaiseau, France
[2] Alma Mater Studiorum - Università di Bologna, Italy
[3] CNRS, LIP, ENS de Lyon, France

Abstract. Wu's positive λ-calculus is a recent call-by-value λ-calculus with sharing coming from Miller and Wu's study of the proof-theoretical concept of focalization. Accattoli and Wu showed that it simplifies a technical aspect of the study of sharing; namely it rules out the recurrent issue of renaming chains, that often causes a quadratic time slowdown. In this paper, we define the natural abstract machine for the positive λ-calculus and show that it suffers from an inefficiency: the quadratic slowdown somehow reappears when analyzing the cost of the machine. We then design an optimized machine for the positive λ-calculus, which we prove efficient. The optimization is based on a new slicing technique which is dual to the standard structure of machine environments.

Keywords: λ-calculus · abstract machines · complexity analyses.

1 Introduction

The λ-calculus is a minimalistic abstract setting that does not come with a fixed implementation schema. This is part of its appeal as a theoretical framework. Yet, for the very same reason, many different implementation techniques have been developed along the decades. How to implement the λ-calculus is in fact a surprisingly rich problem with no absolute best answer.

Recently, a proof theoretical study about focusing by Miller and Wu [28] led to a new λ-calculus with sharing—Wu's positive λ-calculus $\lambda_{\mathtt{pos}}$ [35]—that provides a fresh perspective on, and an improvement of a recurrent efficiency issue as shown by Accattoli and Wu [19]. The present paper studies a somewhat surprising fact related to the efficiency of the positive λ-calculus.

Similar to the ordinary λ-calculus, Wu's calculus is a minimalistic abstract setting. Its sharing mechanism, however, decomposes β-reduction in micro steps and makes it closer to implementations than the ordinary λ-calculus. It is, in fact, almost an abstract machine.

This work stems from the observation that the natural refinement of $\lambda_{\mathtt{pos}}$ as an abstract machine suffers from an inefficiency. Intuitively, the efficiency issue improved by $\lambda_{\mathtt{pos}}$ resurfaces at the lower level of machines. After pointing out the problem, we design an optimized abstract machine and prove it efficient, solving the issue. The adopted *slicing* optimization is new and based on a dual of the standard environment data structure for abstract machines. We believe it to be an interesting new implementation schema.

The Positive λ-Calculus. In [28], Miller and Wu decorate focalized proofs for minimal intuitionistic logic with proof terms. Minimal intuitionistic logic is the proof-theoretical counterpart of the λ-calculus via the Curry-Howard correspondence. Focalization is a technique that constrains the shape of proofs depending on a polarity assignment to atomic formulas. Miller and Wu show that uniformly adopting the negative polarity induces the ordinary λ-calculus, while the positive polarity induces an alternative syntax with (sub-term) sharing, where sharing is represented via let-expressions or, equivalently, explicit substitutions.

An important aspect of this new positive syntax is its *compactness property*, also referred to as *positive sharing*: the shape of terms is highly constrained, more than in most other calculi with let-expressions or explicit substitutions. In particular, applications cannot be nested, arguments can only be variables, applications and abstractions are always shared, and variables cannot be shared. Some of these constraints are also at work in Sabry and Felleisen's *A-normal forms* [31,23], of which positive sharing can be seen as an even more constrained variant; see the introduction of Accattoli and Wu [19] for extended discussions about similar formalisms. Intuitively, the constrained syntax somewhat forces a maximal sharing of sub-terms by ruling out redundant cases of sharing from the grammar for terms.

In [35], Wu endows the positive syntax with rewriting rules. Because of the compact syntax, there is not much freedom for defining the rules: they have to be call-by-value, and they have to be micro-step, that is, the granularity of the substitution process is the one of abstract machines, that replace one variable occurrence at a time. Call-by-name or meta-level substitution on all occurrences of a variable are indeed ruled out because they do not preserve compactness. The outcome is the positive λ-calculus $\lambda_{\mathtt{pos}}$. The difference between $\lambda_{\mathtt{pos}}$ and an abstract machine is only that $\lambda_{\mathtt{pos}}$ does not have rules searching for redexes.

Renaming Chains. In [19], Accattoli and Wu show two things. Firstly, compactness rules out a recurrent issue in the study of sharing and abstract machines, namely *renaming chains*. In general, when one adds to the λ-calculus an explicit substitution construct, noted here $t[x \leftarrow u]$ (equivalent to $\mathsf{let}\ x = u\ \mathsf{in}\ t$, and sharing u for x in t), then one can have chains of shared variables of the following form: $t[x_1 \leftarrow x_2] \dots [x_{n-1} \leftarrow x_n]$.

These renaming, or indirection chains, are a recurrent burden of sharing-based systems, as they lead to both time and space inefficiencies—typically a quadratic time slowdown—and need optimizations to avoid their creations, as done for instance by Sands et al. [32], Wand [34], Friedman et al. [24], and Sestoft [33]. In the literature on sharing and abstract machines, the issue tended to receive little attention, until Accattoli and Sacerdoti Coen focused on it [17]. Removing renaming chains is also essential in the study of reasonable logarithmic space for the λ-calculus, see Accattoli et al. [10].

In the positive λ-calculus, variables cannot be shared. Therefore, renaming chains simply cannot be expressed, ruling out the issue. It is important to point out, however, that forbidding the sharing of variables requires tuning the rewriting rules by adding some meta-level renamings.

Useful Sharing. Secondly, Accattoli and Wu show that the compactness and the removal of renaming chains of the positive λ-calculus have the effect of drastically simplifying *useful sharing*, a sophisticated implementation technique introduced by Accattoli and Dal Lago in the study of reasonable time cost models [9]. Intuitively, useful sharing aims at preventing the useless unfolding of sharing, which is exactly what is achieved by compactness via the restriction of the grammar of terms. That is, positive sharing captures the essence of useful sharing.

To be precise, Accattoli and Wu show that compactness simplifies the *specification* of useful sharing, but they do not provide precise cost analyses—that are an essential aspect of useful sharing—for the positive λ-calculus.

The Inefficiency. The inception of the present work is exactly the desire to develop a cost analysis of $\lambda_{\mathtt{pos}}$. Cost analyses are usually done via a study of an abstract machine for the calculus of interest. The somewhat surprising fact is that the natural abstract machine associated to the positive λ-calculus—first given here, and dubbed Natural POsitive Machine, or *Natural POM*—is inefficient. The culprit is the mentioned meta-level renamings added to compensate for the fact that variables cannot be shared. The implementation of these renamings induces a quadratic (rather than linear) overhead in the number of β-steps. Essentially, the inefficiency of renaming chains reappears at the lower level of the Natural POM even if the chains themselves have disappeared.

Useful sharing is meant to reduce the overhead from exponential to polynomial, so the inefficiency does not invalidate the value of $\lambda_{\mathtt{pos}}$ for specifying useful sharing. The literature however contains abstract machines for useful sharing having only a *linear* overhead (by Accattoli and co-authors [16,13,7,8]), thus the Natural POM compares poorly, despite the fact that the positive λ-calculus is a better specification of useful sharing than those in the literature.

The Solution: Sub-Term Property and Slices. We then design an optimized abstract machine, the *Sliced POM*, that solves the issue and recovers linearity in the number of β-steps. To give a hint of the solution, we need to say a bit more about the problem. Complexity analyses of abstract machines are based on their crucial *sub-term property*, stating that all the terms duplicated along the run of the machine are sub-terms of the initial term. The property allows one to bound the cost of each transition using the initial term, which is essential in order to express the cost of the run as a function of the size of the initial term.

Now, the Natural POM does verify the sub-term property *with respect to duplications*; the source of the inefficiency for once is not the duplication process. The source is nonetheless related to a lack of sub-term property *for renamings*: the additional meta-level renamings act over a scope that might not be a sub-term of the initial term. The solution amounts to *slicing* such scopes in slices that are sub-terms of the initial term, and in noting that each meta-level renaming is always confined to exactly one slice. The slicing technique is managed very easily, via an additional basic data structure, the slice stack.

Slices vs Environments. A pleasant aspect is that the slicing stack can be seen as *dual* to the environment data structure used to manage sharing:

- an environment entry $[x \leftarrow t]$ stores the delayed substitution of t for x waiting for the evaluation of active term to expose occurrences of x to replace;
- an entry $t[x \leftarrow \cdot]$ of the slice stack waits for the active term to become a variable z to be substituted for x in the slice t.

This duality suggests that the slicing technique exposes a new natural structure. The meta-level renamings of the calculus are similar to the ones generated by β-redexes. In traditional settings with sharing, indeed, the optimizations to remove renaming chains act on the environment. We find it interesting that positive sharing *disentangles* meta-level renamings from the usual substitution process, and manages them via a sort of dual mechanism.

On the Value of Positive Sharing. The reader might wonder what the value of positive sharing is, given that the issue that it is supposed to solve does reappear at the level of machines and forces the design of a new solution. In our opinion, the value of positive sharing is in re-structuring the study of sharing techniques. The study by Accattoli and Wu [19] suggests that positive sharing *removes* the issue of renaming chains, but our work provides a refined picture. Namely, positive sharing enables a neater theory of sharing, disentangling renaming chains from the treatment of explicit substitutions, and encapsulating their inefficiency at the lower level of implementations choices. Additionally, it brings to the fore the concept of slice stack, which we believe is an interesting addition to the theory of implementations of the λ-calculus.

OCaml Implementation. For lack of space we overview in Appendix A of the technical report [18] an implementation in OCaml of the Sliced POM, to be found at https://github.com/sacerdot/PositiveAbstractMachine. The implementation is meant to provide further evidence on the cost of the atomic operations of the machine. Moreover, it allows the interested reader to enter ordinary λ-terms and see how they are transformed into positive terms (via the transformation studied by Accattoli and Wu in [19]) and then run by the Sliced POM.

The implementation is a prototype: it does not attempt to further optimize space usage or to recover garbage memory, nor does it optimize the code that is unrelated to running the machine (*e.g.* the pretty-printing of machine states).

Proofs. Most proofs are omitted. They can be found in the appendix of the technical report on arXiv [18].

2 The Positive λ-Calculus

In this section, we present Wu's positive λ-calculus [35]. Precisely, we adopt the *explicit open* variant from Accattoli and Wu [19]. For simplicity, we simply refer to it as the positive λ-calculus, and note it λ_{pos}. We depart slightly from the

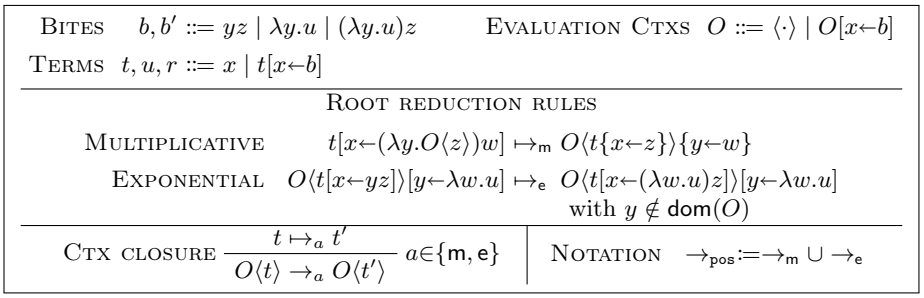

Fig. 1. The positive λ-calculus λ_{pos}.

presentation in [19], omitting the garbage collection rule because it can always be postponed, as shown in [19], and changing some notations. The definition is in Fig. 1.

Terms. The positive λ-calculus uses let-expressions, similarly to Moggi's CbV calculus [29,30]. We do however write a let-expression $\mathsf{let}\ x = u\ \mathsf{in}\ t$ as a more compact *explicit substitution* $t[x\leftarrow u]$ (ES for short), which binds x in t. Moreover, for the moment our let/ES does not fix an order of evaluation between t and u, in contrast to many papers in the literature (*e.g.* Levy et al. [27]) where u is evaluated first. The evaluation order shall be fixed in the next section.

While we borrow the terminology *explicit substitution* from the seminal work of Abadi et al. [1], the way we employ the construct is deeply different, since the theory of ESs has progressed considerably since that first work. In particular, we do use variable names instead of de Bruijn indices, and our ESs do not move through the structure of the term, instead they act *at a distance* (explained below). The positive λ-calculus is rather reminiscent of Sabry and Felleisen's *A-normal forms* [31,23], or of the calculi for call-by-need by Lunchbury [26] and Sestoft [33], of which it can be thought as a more constrained variant.

In fact, ESs are here used in a slightly unusual way even with respect to more recent work on ESs. In λ_{pos}, as in the λ-calculus, there are only three constructors, variables, applications, and abstractions. There are however, various differences, namely:

- Applications have either shape yz or $(\lambda y.t)z$, that is, arguments can only be variables and the left sub-term cannot be an application;
- Applications and abstractions are dubbed *bites* (following the terminology of Accattoli et al. [7]) and are always shared, that is, standalone applications and abstractions are not allowed by the grammar. They can only be introduced by the ESs constructs $[x\leftarrow yz]$, $[x\leftarrow(\lambda y.t)z]$, and $[x\leftarrow\lambda y.u]$;
- Positive sharing is peculiar as positive terms are *not* shared in general, that is, $t[x\leftarrow u]$ is not a positive term. There is no construct for sharing variables or applications/abstractions with top-level sharing, *i.e.* $t[x\leftarrow y]$ or $t[x\leftarrow yz[y\leftarrow\lambda w.u]]$ are not terms of λ_{pos}. In particular, the absence of $t[x\leftarrow y]$ is what forbids renaming chains.

The set of free variables of a term t is denoted by $\mathsf{fv}(t)$ and it is defined as expected. Terms are identified up to α-renaming. We use $t\{x\leftarrow y\}$ for the capture-avoiding substitution of y for each free occurrence of x in t; in this paper we never need the more general operation $t\{x\leftarrow u\}$ substituting terms. The meta-level renamings mentioned in the introduction, to be used to compensate for the absence of $t[x\leftarrow y]$, shall be instances of $t\{x\leftarrow y\}$.

Open Setting. Evaluation in λ_{pos} shall be open in the sense promoted by Accattoli and Guerrieri [12], that is, it does not go under abstraction (also referred to as *weak*) and terms *can* be open (but do not have to). As discussed at length by Accattoli and Guerrieri, this is a more general framework than the standard for functional programming languages of weak evaluation and *closed* terms. The increased generality enables the developed theory to scale up to evaluation under abstraction, which is needed to model proof assistants, by iterating evaluation under abstraction—this is done for instance by Grégoire and Leroy for Coq [25]—because the body of abstractions cannot be assumed to be closed. See [12] for more discussions.

Open Contexts. Contexts are terms with exactly one occurrence of the *hole* $\langle\cdot\rangle$, an additional constant, standing for a removed sub-term. We shall use various notions of contexts. The most general ones in this paper are *open contexts* O, that are simply lists of ESs. The main operation about contexts is *plugging* $O\langle t\rangle$ where the hole $\langle\cdot\rangle$ in context O is replaced by t. Plugging, as usual with contexts, can capture variables—for instance $(\langle\cdot\rangle[x\leftarrow b])\langle x\rangle = x[x\leftarrow u]$.

The domain $\mathsf{dom}(O)$ of a context is the set of variables possibly captured by O (i.e., on which O has an ES scoping over $\langle\cdot\rangle$); example: setting $O := \langle\cdot\rangle[x\leftarrow\lambda y.y[y'\leftarrow y''y'']][z\leftarrow z'z'']$ one obtains $\mathsf{dom}(O) = \{x, z\}$. When $x \in \mathsf{dom}(O)$, we also use the notation $O(x)$ to denote the bite associated to x in O; in the example, $O(z) = z'z''$.

As it is immediately seen from the grammar of positive terms, every positive term t can be written uniquely as $O\langle x\rangle$ for some x and O, with O possibly capturing x. If $t = O\langle x\rangle$ then x is referred to as the *head variable* of t.

Rewriting Rules. There are two rewriting rules, following the *at a distance* style promoted by Accattoli and Kesner [14], which involves contexts in the definition of the rules, even before the contextual closure. The rules names come from the connection with linear logic proof nets, which is omitted here.

The multiplicative rule $\to_{\mathsf{m}}$ reduces a shared β-redex. The rule is forced to decompose the body of the abstraction as $O\langle z\rangle$ in order to write the reduct. The point is that the simpler rule $t[x\leftarrow(\lambda y.u)w] \mapsto_{\mathsf{m}} t[x\leftarrow u[y\leftarrow w]]$ does not respect the grammars of the positive λ-calculus, since ESs such as $[x\leftarrow u]$ and $[y\leftarrow w]$, as well as their nesting, are forbidden. Let us show an example of multiplicative step stressing the action of renamings:

$$\begin{array}{lcl} z[x\leftarrow yz][z\leftarrow(\lambda w.z'[x'\leftarrow wz'])y'] & \to_{\mathsf{m}} & z[x\leftarrow yz]\{z\leftarrow z'\}[x'\leftarrow wz']\{w\leftarrow y'\} \\ & = & z'[x\leftarrow yz'][x'\leftarrow y'z'] \end{array}$$

The exponential rule $\to_{\mathsf{e}}$ simply replaces an applied variable with the associated abstraction. Note that arguments, that are variables, are never replaced, because their replacement would—once more—step out of the grammar of the positive λ-calculus.

Both rules are closed by open contexts and together form the rewriting relation $\to_{\mathtt{pos}}$ of the positive λ-calculus.

Translating λ-Terms. In [19], Accattoli and Wu show how to translate λ-terms to positive terms in a way that induces a simulation of call-by-value evaluation. We refer the interested reader to their work, because in this paper we only deal with positive terms.

Diamond. The defined calculus is non-deterministic. Consider for instance $t := z[x{\leftarrow}yy][z{\leftarrow}(\lambda w.w)y'][y{\leftarrow}\lambda x'.u]$. One has for instance the following diagram:

$$\begin{array}{ccc} t & \xrightarrow{\quad\mathsf{m}\quad} & y'[x{\leftarrow}yy][y{\leftarrow}\lambda x'.u] \\ {\scriptstyle\mathsf{e}}\downarrow & & \vdots{\scriptstyle\mathsf{e}} \\ z[x{\leftarrow}(\lambda x'.u)y][z{\leftarrow}(\lambda w.w)y'][y{\leftarrow}\lambda x'.u] & \overset{\mathsf{m}}{\cdots\cdots\to} & y'[x{\leftarrow}(\lambda x'.u)y][y{\leftarrow}\lambda x'.u] \end{array}$$

The calculus however is confluent, and even more than confluent, it has the diamond property. According to Dal Lago and Martini [20], a relation $\to$ is *diamond* if $u_1 \leftarrow t \to u_2$ imply $u_1 = u_2$ or $u_1 \to r \leftarrow u_2$ for some r. The diamond property expresses a relaxed form of determinism, since it states that different choices cannot change the result *nor the length of evaluation sequences* (note that the diagram closes in either zero or one steps on both sides).

Theorem 1 (Positive diamond, [19]). *Relation $\to_{\mathtt{pos}}$ is diamond.*

Sub-Term Property. When the substitution process is decomposed in micro steps, usually it is possible to bound the cost of each duplication along an evaluation sequence using the size of the initial term of the sequence—this is the sub-term property. It is crucial in order to analyze the cost of evaluation sequences as a function of the size of the initial term and the number of steps, since the main danger of excessive cost usually comes from duplications. The property does not hold, for instance, for the ordinary λ-calculus (see Accattoli [2, Section 3]) that relies on meta-level (rather than micro-step) substitution. The positive λ-calculus has the sub-term property, that is expressed for values, because they are what is duplicated by the exponential rule.

Lemma 1 (Sub-term property). *Let $t \to_{\mathtt{pos}}^{*} u$ be a reduction sequence. Then $|\lambda x.r| \leq |t|$ for every bite $\lambda x.r$ duplicated by a e-step of the sequence.*

Proof. Formally, the proof is a straightforward induction on the length of the reduction sequence, by looking at the last step and using the *i.h.* The following informal observations however are probably enough. Evaluation duplicates variables (in $\to_{\mathsf{m}}$) and abstractions (in $\to_{\mathsf{e}}$). The substitution of variables cannot change the size of abstractions. For abstractions, note that replaced variables are out of abstractions, and open contexts never enter abstractions, so that all abstractions of the sequence can be traced back to t (up to α-renaming). □

3 The Right Strategy

Usually, abstract machines are deterministic and implement a deterministic evaluation strategy. Therefore, in this section, we define a deterministic strategy for the positive λ-calculus and prove its basic properties.

We adopt the right(-to-left) strategy $\to_{\mathtt{r}}$ that picks redexes from right to left. It is a standard approach but it turns out that defining it in $\lambda_{\mathtt{pos}}$ is a bit tricky, because of e-steps, where two ESs interact at a distance.

Redex Positions. For calculi at a distance, the notion of position of a redex—which is mandatory to determine the rightmost redex—might not be as expected. For us, a position in a term is simply an open context.

Definition 1 (Redex positions). *When taking into account the contextual closure,* m*-steps and* e*-steps have the following shapes:*

$$\begin{array}{rcccl} t & = & O'\langle r[x{\leftarrow}(\lambda y.O\langle z\rangle)w]\rangle & \to_{\mathsf{m}} & O'\langle O\langle r\{x{\leftarrow}z\}\rangle\{y{\leftarrow}w\}\rangle = u \\ t & = & O'\langle O\langle r[x{\leftarrow}yz]\rangle[y{\leftarrow}\lambda w.q]\rangle & \to_{\mathsf{e}} & O'\langle O\langle r[x{\leftarrow}(\lambda w.q)z]\rangle[y{\leftarrow}\lambda w.q]\rangle = u \end{array}$$

The position of the m*-step is simply given by the surrounding context* O'*. The position of the* e*-step is the context* $O'\langle O[y{\leftarrow}\lambda w.q]\rangle$*.*

The rationale behind the position of e-steps is the idea that they are triggered when one finds the variable to substitute, and not when one finds the abstraction—this is indeed how abstract machines work. This approach is standard in the literature about ESs at a distance, see *e.g.* Accattoli et al. [6].

Example: in $x[x{\leftarrow}yz][x'{\leftarrow}y'z][y'{\leftarrow}\lambda w'.r][y{\leftarrow}\lambda w.u]$ there are two redexes (on y and y') and the rightmost one is on y', despite the ES on y' occurring to the left of the one on y.

Right Contexts. We specify the strategy via the notion of right contexts, itself specified via the notion of applied free variable. In fact, there are two dual ways of defining right contexts. We present them both and prove their equivalence.

Definition 2 (Applied free variables, right contexts). *The set of applied (and out of abstraction bodies) free variables* $\mathsf{afv}(O)$ *of an open context* O *is defined as:*

$$\textsc{Set of applied free variables}$$

$$\begin{array}{rcl|rcl} \mathsf{afv}(\langle\cdot\rangle) & \coloneqq & \emptyset & \mathsf{afv}(O[x{\leftarrow}yz]) & \coloneqq & (\mathsf{afv}(O)\setminus\{x\})\cup\{y\} \\ \mathsf{afv}(O[x{\leftarrow}(\lambda y.t)z]) & \coloneqq & \mathsf{afv}(O)\setminus\{x\} & \mathsf{afv}(O[x{\leftarrow}\lambda y.t]) & \coloneqq & \mathsf{afv}(O)\setminus\{x\} \end{array}$$

The two definitions of right contexts (we use on purpose the same meta-variable, since they shall be proved equivalent right next) are given by:

$$\textsc{Outside-in right contexts}$$

$$R ::= \langle\cdot\rangle \mid R[x{\leftarrow}yz] \mid R[x{\leftarrow}\lambda y.u] \text{ if } x\notin\mathsf{afv}(R)$$

$$\textsc{Inside-out right contexts}$$

$$R ::= \langle\cdot\rangle \mid R\langle\langle\cdot\rangle[x{\leftarrow}yz]\rangle \text{ if } R(y)\neq\lambda w.t \mid R\langle\langle\cdot\rangle[x{\leftarrow}\lambda y.u]\rangle$$

TRANSITIONS

ACTIVE CODE	RIGHT CTX		ACTIVE CODE	RIGHT CTX	
$t[x\leftarrow\lambda y.u]$	$\lhd R$	$\rightsquigarrow_{\mathtt{sea}_1}$	t	$\lhd R\langle\langle\cdot\rangle[x\leftarrow\lambda y.u]\rangle$	
$t[x\leftarrow yz]$	$\lhd R$	$\rightsquigarrow_{\mathtt{sea}_2}$	t	$\lhd R\langle\langle\cdot\rangle[x\leftarrow yz]\rangle$	$(*)$
$t[x\leftarrow yz]$	$\lhd R$	$\rightsquigarrow_{\mathtt{e}}$	$t[x\leftarrow(\lambda w.u)^{\alpha}z]$	$\lhd R$	$(\#)$
$t[x\leftarrow(\lambda y.O\langle z\rangle)w]$	$\lhd R$	$\rightsquigarrow_{\mathtt{m}}$	$O\langle t\{x\leftarrow z\}\rangle\{y\leftarrow w\}$	$\lhd R$	

$(*)$ if $y\notin\mathsf{dom}(R)$ or $R(y)\neq\lambda w.u$; $(\#)$ if $R(y)=\lambda w.u$.

Fig. 2. The Natural Positive Machine (Natural POM).

Lemma 2. *Outside-in and inside-out right contexts coincide.*

Because of the lemma, we only speak of right contexts, and adopt the most convenient definition in each case.

Right Strategy. We now have all the ingredients to define the right strategy and prove its basic properties.

Definition 3 (Right strategy). *A $\rightarrow_{\mathtt{pos}}$ step is* right *when its position is a right context. The right strategy $\rightarrow_{\mathtt{r}}$ reduces at each step a right redex, if any. We write $t\rightarrow_{\mathsf{rm}}u$ (resp. $t\rightarrow_{\mathsf{re}}u$) for a right* m*-step (resp.* e*-step).*

Lemma 3 (Basic properties of the right strategy).

1. Determinism*: if $t\rightarrow_{\mathtt{r}}t_1$ and $t\rightarrow_{\mathtt{r}}t_2$ then $t_1=t_2$;*
2. No premature stops*: if $t\rightarrow_{\mathtt{pos}}u$ then $t\rightarrow_{\mathtt{r}}r$ for some r.*

4 A Natural but Inefficient Positive Machine

In this section, we give the natural abstract machine implementing the right strategy of the previous section, obtained by adding a basic mechanism for searching redexes, discuss the (in)efficiency of the machine, and explain how to modify it as to make it efficient. The tone is slightly informal. The next sections shall formally define and study the modified machine.

Typical abstract machines for the λ-calculus in the literature are the Krivine abstract machine (KAM) or Felleisen and Friedman's CEK machine [22] that use many environments, closures, and never α-rename. Here we rely on a different and simpler approach, having only one global environment (represented as a context), no closures, and using α-renaming. For comparisons between the two approaches, see Accattoli and Barras [5].

The Natural POM. States of the Natural POsitive Machine (Natural POM[4]) are pairs $t\lhd R$ denoting a pointer $\lhd$ inside the structure of the term $u:=R\langle t\rangle$

[4] Acronyms for abstract machines tend to end with AM for Abstract Machine. We avoided PAM, however, because there already is a PAM (Pointer Abstract Machine) in the literature, introduced by Danos et al. [21].

represented by the state. The pointer represents the current position of the machine over u. Initially, the pointer is at the rightmost position, that is, initial states have shape $t \lhd \langle\cdot\rangle$. The Natural POM has the four transitions in Fig. 2. The differences with respect to the positive λ-calculus are that:

- *Search*: there are two search transitions $\mathtt{sea}_1$ and $\mathtt{sea}_2$ to move the pointer, in order to search for redexes, that move $\lhd$ right-to-left (hence the symbol);
- *Names*: there is a more controlled management of α-conversion, which is performed only on the exponential step, when copying abstractions.

Transition $\mathtt{sea}_1$ moves the abstraction from the active code to the right context. Transition $\mathtt{sea}_2$ moves $[x{\leftarrow}yz]$ when y is a free variable (that is, $y \notin \mathsf{dom}(R)$) or when it is bound by an ES in R but not one that contains an abstraction, so that the application yz does not give rise to a multiplicative redex. Note that these two cases are exactly the inside-out definition of right contexts in Def. 2.

Example of Run. As an example of execution of the Natural POM, we consider the first few transitions of the infinite run for the positive representation $x[x{\leftarrow}yy][y{\leftarrow}\lambda z.w[w{\leftarrow}zz]]$ of the paradigmatic looping λ-term $\Omega \coloneqq (\lambda y.yy)(\lambda z.zz)$:

Active code		Right ctx		
$x[x{\leftarrow}yy][y{\leftarrow}\lambda z.w[w{\leftarrow}zz]]$	$\lhd$	$\langle\cdot\rangle$	$\rightsquigarrow_{\mathtt{sea}_1}$	
$x[x{\leftarrow}yy]$	$\lhd$	$[y{\leftarrow}\lambda z.w[w{\leftarrow}zz]]$	$\rightsquigarrow_{\mathtt{e}}$	
$x[x{\leftarrow}(\lambda z'.w'[w'{\leftarrow}z'z'])y]$	$\lhd$	$[y{\leftarrow}\lambda z.w[x{\leftarrow}zz]]$	$\rightsquigarrow_{\mathtt{m}}$	
$w'[w'{\leftarrow}yy]$	$\lhd$	$[y{\leftarrow}\lambda z.w[x{\leftarrow}zz]]$	$\rightsquigarrow_{\mathtt{e}}$	$\ldots$

Basics of the Complexity of Abstract Machines. The problem with the Natural POM concerns the complexity of its overhead with respect to the calculus. Let us first recall the basics of the topic. For abstract machines, the complexity of the overhead for a machine run r of initial term t is measured with respect to two parameters: the number of steps of the underlying calculus and the size $|t|$ of the initial term. A large number of machines has complexity linear in both parameters—shortened to *bi-linear*—as first shown by Accattoli et al. [3], and repeatedly verified after that for even more machines [16,4,7,13,8]. The bi-linearity of the overhead then becomes a design principle, or, when it fails, a strong indication of an inefficiency, and that the machine can be improved.

The key property enabling complexity analyses of machines is the sub-term property already discussed for $\lambda_{\mathtt{pos}}$. It ensures that the size of states cannot grow more than the size of the initial term at each transition. In turn, this fact usually implies that the time cost is also linearly bounded.

The Inefficiency of the Natural POM. The natural POM inherits the sub-term property from the calculus. Its inefficiency is related to the multiplicative transition $\rightsquigarrow_{\mathtt{m}}$, namely to the meta-level renaming $t\{x{\leftarrow}z\}$ (referring to Fig. 2). Meta-level substitutions are possibly costly. Usually, the danger is the time spent in making copies of the term to duplicate, which might be big and/or because

many copies of it might be required. The size of the term to duplicate is not the problem here, since the involved term is a simple variable z.

The culprit actually is the *size of the scope* over which the renaming can take place, rather than the number of copies. There are two renamings. The renaming $\{y \leftarrow w\}$ is harmless: its scope seems to be $O\langle t\{x \leftarrow z\}\rangle$ but in fact y can occur only in $O\langle z\rangle$ (which is the body of the abstraction of y in the source state of the transition), and the sub-term property guarantees that the size of $O\langle z\rangle$ is bound by the size of the initial term. Therefore, one can propagate $\{y \leftarrow w\}$ on $O\langle z\rangle$ before computing the full reduct, staying within a linear cost.

For the renaming $t\{x \leftarrow z\}$, however, there is in general no connection between t and the initial term. For the first multiplicative step of the run, t is a sub-term of the initial term. But the step itself re-combines O and t as to create a new term unrelated to the initial one. Consider for instance the following run, where we assume that O captures z and that $(\lambda y.(O\langle z\rangle[z' \leftarrow b]))^{\alpha} = \lambda y'.(O'\langle z'\rangle[z'' \leftarrow b'])$:

$$
\begin{array}{lll}
 & \textsc{Active code} & \textsc{Right ctx} \\
\hline
 & t[x \leftarrow x'w][x' \leftarrow \lambda y.(O\langle z\rangle[z' \leftarrow b])] & \lhd\ \langle\cdot\rangle \\
\rightsquigarrow_{\mathtt{sea}_1} & t[x \leftarrow x'w] & \lhd\ [x' \leftarrow \lambda y.(O\langle z\rangle[z' \leftarrow b])] \\
\rightsquigarrow_{\mathtt{e}} & t[x \leftarrow (\lambda y'.(O'\langle z'\rangle[z'' \leftarrow b']))w] & \lhd\ [x' \leftarrow \lambda y.(O\langle z\rangle[z' \leftarrow b])] \\
\rightsquigarrow_{\mathtt{m}} & O'\langle t\{x \leftarrow z'\}\rangle[z'' \leftarrow b']\{y' \leftarrow w\} & \lhd\ [x' \leftarrow \lambda y.(O\langle z\rangle[z' \leftarrow b])]
\end{array}
\tag{1}
$$

Now, if the bite b' is a β-redex then the next transition is multiplicative and it is going to rename over $O'\langle t\{x \leftarrow z'\}\rangle$ which is not a sub-term of the initial term.

How Big is the Inefficiency? When the time cost depends on something that is not a sub-term of the initial term, things can easily escalate up to exponential costs, as in the paradigmatic case of size explosion, see Accattoli [2, Section 3]. Luckily, here things do not go sideways, there is only a mild inefficiency. The key point is that the sub-term property for duplications *does hold*: it ensures that the size of the whole state grows bi-linearly with the number of transitions, so there is no exponential growth. The problematic renaming—and thus each multiplicative transition—might then have to scan a scope that is at worst bi-linear in the length of the preceding run and the size of the initial term. A standard argument then gives a quadratic bound (in the number of steps and the size of the initial term) on the global cost of all multiplicative steps.

Let us give an example. We use a diverging term because it is the simplest example showcasing the phenomenon. We define the positive representation of $\Omega_3 \coloneqq \delta_3\delta_3$ where $\delta_3 \coloneqq \lambda x.x(xx)$. Let $\tau_3 \coloneqq x[x \leftarrow yz][z \leftarrow yy]$ and let $\tau_3' \coloneqq x'[x' \leftarrow y'z'][z' \leftarrow y'y']$ and so on. The analogous of Ω_3 is $\tau_3[y \leftarrow \lambda y.\tau_3]$, that runs as follows:

Active code	Right ctx	
$x[x{\leftarrow}yz][z{\leftarrow}yy][y{\leftarrow}\lambda y.\tau_3]$	$\lhd\ \langle\cdot\rangle$	$\rightsquigarrow_{\mathtt{sea}_1}$
$x[x{\leftarrow}yz][z{\leftarrow}yy]$	$\lhd\ [y{\leftarrow}\lambda y.\tau_3]$	$\rightsquigarrow_{\mathtt{e}}$
$x[x{\leftarrow}yz][z{\leftarrow}(\lambda y'.\tau_3')y]$	$\lhd\ [y{\leftarrow}\lambda y.\tau_3]$	$=$
$x[x{\leftarrow}yz][z{\leftarrow}(\lambda y'.x'[x'{\leftarrow}y'z'][z'{\leftarrow}y'y'])y]$	$\lhd\ [y{\leftarrow}\lambda y.\tau_3]$	$\rightsquigarrow_{\mathtt{m}}$
$x[x{\leftarrow}yx'][x'{\leftarrow}yz'][z'{\leftarrow}yy]$	$\lhd\ [y{\leftarrow}\lambda y.\tau_3]$	$\rightsquigarrow_{\mathtt{e}}$
$x[x{\leftarrow}yx'][x'{\leftarrow}yz'][z'{\leftarrow}(\lambda y''.\tau_3'')y]$	$\lhd\ [y{\leftarrow}\lambda y.\tau_3]$	$\rightsquigarrow_{\mathtt{m}}$
$x[x{\leftarrow}yx'][x'{\leftarrow}yx''][x''{\leftarrow}yz''][z''{\leftarrow}yy]$	$\lhd\ [y{\leftarrow}\lambda y.\tau_3]$	$\rightsquigarrow_{\mathtt{e}}$ $\quad\ldots$

It is clear that the scopes of the renamings keep growing, even if the number of renamed occurrences by each renaming is constant.

Removing the Inefficiency: Slices. An observation about the run (1) above suggests how to improve the situation. The idea is to delay the merging of t and $O'\langle z'\rangle$ as $O'\langle t\{x{\leftarrow}z'\}\rangle$ (similarly to how ESs delay meta-level substitutions) by putting the pair (t, x)—dubbed *slice* and that shall actually be denoted with $t[x{\leftarrow}\cdot]$—in a new stack storing delayed merges, and keeping as active code $O'\langle z'\rangle$. The observation is that if b' is a β-redex and generates a renaming of z'' then one only needs to inspect $O'\langle z'\rangle$—which, crucially, is a sub-term of the initial term—because z'' cannot occur in t, given that it comes from the body of an abstraction out of t.

Before defining the Sliced POM, and prove that slices do solve the problem, we start over with a more formal approach to abstract machines.

5 Preliminaries About Abstract Machines

In this section, we fix the terminology about abstract machines. We follow Accattoli and co-authors [16,4,7,13,8], adapting their notions to our framework. We mostly stay abstract. In the next section, we shall instantiate the abstract notions on a specific machine.

Abstract Machines Glossary. Abstract machines manipulate *pre-terms*, that is, terms without implicit α-renaming. In this paper, an *abstract machine* is a quadruple $\mathsf{M} = (\mathtt{States}, \rightsquigarrow, \cdot \lhd \cdot, \bar{\cdot})$ the components of which are as follows.

- *States.* A state $\mathtt{Q} \in \mathtt{States}$ is composed by the *active term* t, plus some data structures; the machine of the next section shall have two data structures. Terms in states are actually pre-terms.
- *Transitions.* The pair $(\mathtt{States}, \rightsquigarrow)$ is a transition system with transitions $\rightsquigarrow$ partitioned into *principal transitions*, whose union is noted $\rightsquigarrow_{\mathtt{pr}}$ and that are meant to correspond to rewriting steps on the calculus, and *search transitions*, whose union is noted $\rightsquigarrow_{\mathtt{sea}}$, that take care of searching for (principal) redexes.
- *Initialization.* The component $\lhd \subseteq \Lambda \times \mathtt{States}$ is the *initialization relation* associating terms to initial states. It is a *relation* and not a function because

$t \triangleleft \mathtt{Q}$ maps a term t (considered modulo α) to a state $\mathtt{Q}$ having a *pre-term representant* of t (which is not modulo α) as active term. Intuitively, any two states $\mathtt{Q}$ and $\mathtt{Q}'$ such that $t \triangleleft \mathtt{Q}$ and $t \triangleleft \mathtt{Q}'$ are α-equivalent. A state $\mathtt{Q}$ is *reachable* if it can be reached starting from an initial state, that is, if $\mathtt{Q}' \rightsquigarrow^* \mathtt{Q}$ where $t \triangleleft \mathtt{Q}'$ for some t and $\mathtt{Q}'$, shortened as $t \triangleleft \mathtt{Q}' \rightsquigarrow^* \mathtt{Q}$.
- *Read-back.* The read-back function $\bar{\cdot} : \mathtt{States} \to \Lambda$ turns reachable states into terms and satisfies the *initialization constraint*: if $t \triangleleft \mathtt{Q}$ then $\overline{\mathtt{Q}} =_\alpha t$.

Further Terminology and Notations. A state is *final* if no transitions apply. A *run* $r : \mathtt{Q} \rightsquigarrow^* \mathtt{Q}'$ is a possibly empty finite sequence of transitions, the length of which is noted $|r|$; note that the first and the last states of a run are not necessarily initial and final. If a and b are transitions labels (that is, $\rightsquigarrow_a \subseteq \rightsquigarrow$ and $\rightsquigarrow_b \subseteq \rightsquigarrow$) then $\rightsquigarrow_{a,b} := \rightsquigarrow_a \cup \rightsquigarrow_b$ and $|r|_a$ is the number of a transitions in r.

Well-Boundness and Renamings. For the machine in this paper, the pre-terms in initial states shall be *well-bound*, that is, they have pairwise distinct bound names; for instance $w[w \leftarrow \lambda z.z][x \leftarrow \lambda y.y]$ is well-bound while $w[w \leftarrow \lambda y.y][x \leftarrow \lambda y.y]$ is not. We shall also write t^α in a state $\mathtt{Q}$ for a *fresh well-bound renaming* of t, *i.e.* t^α is α-equivalent to t, well-bound, and its bound variables are fresh with respect to those in t and in the other components of $\mathtt{Q}$.

Mechanical Bismulations. Machines are usually showed to be correct with respect to a strategy via some form of bisimulation relating terms and states. The notion that we adopt is here dubbed *mechanical bisimulation*. The definition, tuned towards complexity analyses, requires a perfect match between the steps of the evaluation sequence and the principal transitions of the machine run.

Definition 4 (Mechanical bisimulation). *A machine* $\mathsf{M} = (\mathtt{States}, \rightsquigarrow, \cdot \triangleleft \cdot, \bar{\cdot})$ *and a strategy* $\rightarrow_{\mathtt{str}}$ *on terms are mechanical bisimilar when, given an initial state* $t \triangleleft \mathtt{Q}$*:*

1. Runs to evaluations*: for any run* $r : t \triangleleft \mathtt{Q} \rightsquigarrow^* \mathtt{Q}'$ *there exists an evaluation* $e : t \rightarrow^*_{\mathtt{str}} \overline{\mathtt{Q}'}$*;*
2. Evaluations to runs*: for every evaluation* $e : t \rightarrow^*_{\mathtt{str}} u$ *there exists a run* $r : t \triangleleft \mathtt{Q} \rightsquigarrow^* \mathtt{Q}'$ *such that* $\overline{\mathtt{Q}'} = u$*;*
3. Principal matching*: for every principal transition* $\rightsquigarrow_{\mathtt{a}}$ *of label* a *of* M*, in both previous points the number* $|r|_{\mathtt{a}}$ *of* $\mathtt{a}$*-transitions in* r *is exactly the number* $|e|_a$ *of of* $\mathtt{a}$*-steps in the evaluation* e*, i.e.* $|e|_{\mathtt{a}} = |r|_{\mathtt{a}}$*.*

The proof that a machine and a strategy are in a mechanical bisimulation follows from some basic properties, grouped under the notion of *distillery*, following Accattoli et al. [3] (but removing their use of structural equivalence, that here is not needed). The intuition behind the notion of distillery is that the calculus *distils* the machine in that it removes the search mechanism—as captured by the search transparency property below—as well as the organization of states in data structures, while it faithfully mimics the underlying dynamics, as captured by the principal projection and halt properties. At the meta-level, the technique

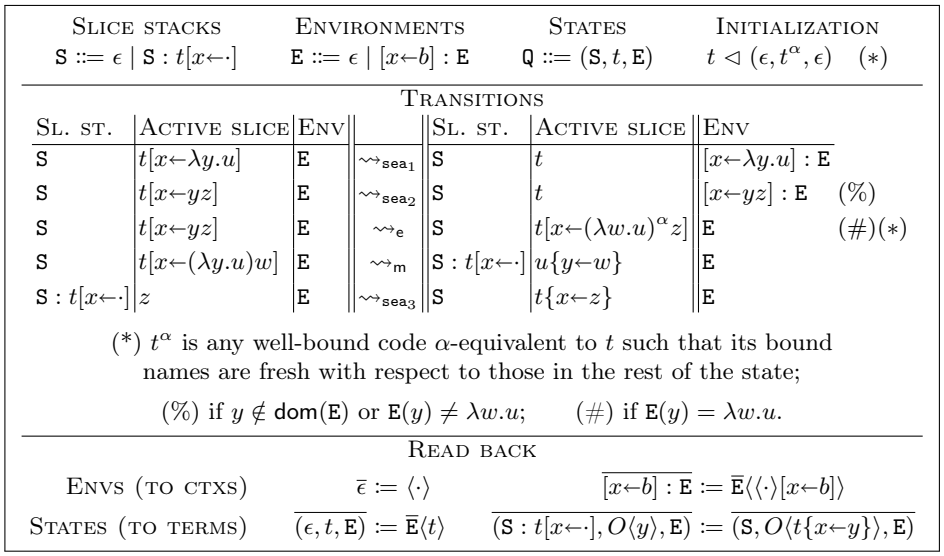

Fig. 3. The Sliced Positive Machine (Sliced POM).

is also meant to distil the reasoning behind bisimulation proofs: the mechanical bisimulation theorem after the definition is proved abstractly by relying solely on the properties defining a distillery.

Definition 5 (Distillery). *A machine* $\mathsf{M} = (\mathtt{States}, \rightsquigarrow, \cdot \triangleleft \cdot, \bar{\cdot})$ *and a strategy* $\to_{\mathtt{str}}$ *are a* distillery *if the following conditions hold:*

1. Principal projection*:* $\mathtt{Q} \rightsquigarrow_{\mathtt{a}} \mathtt{Q}'$ *implies* $\overline{\mathtt{Q}} \to_{\mathtt{a}} \overline{\mathtt{Q}'}$ *for every principal transition of label* $\mathtt{a}$*;*
2. Search transparency*:* $\mathtt{Q} \rightsquigarrow_{\mathtt{sea}} \mathtt{Q}'$ *implies* $\overline{\mathtt{Q}} = \overline{\mathtt{Q}'}$*;*
3. Search transitions terminate*:* $\rightsquigarrow_{\mathtt{sea}}$ *terminates;*
4. Determinism*:* $\to_{\mathtt{str}}$ *is deterministic;*
5. Halt*:* M *final states decode to* $\to_{\mathtt{str}}$*-normal terms.*

Theorem 2 (Sufficient condition for mechanical bisimulations). *Let a machine* M *and a strategy* $\to_{\mathtt{str}}$ *be a distillery. Then, they are mechanical bisimilar.*

6 The Sliced Positive Machine

In this section, we present the optimized machine for the positive λ-calculus and establish the bisimulation between the new machine and the right strategy $\to_{\mathtt{r}}$.

The Sliced POM: Data Structures. The Sliced POsitive Machine (Sliced POM) is defined in Fig. 3. The machine has two data structures, the environment $\mathtt{E}$ and the slice stack $\mathtt{S}$. The active code is re-dubbed *active slice*.

Environments $\mathtt{E}$ are nothing else but encodings of right contexts R. We prefer to change the representation for two reasons. Firstly, it is closer to how contexts are represented in the OCaml implementation. Secondly, it is a bit more precise: environments are free structures, and an invariant shall prove that they decode to right contexts (which are defined via some additional conditions).

The slice stack is simply a list of slices $t[x\leftarrow\cdot]$. Note the duality between environments and slice stacks: the entries of both are pairs of a term t and a variable x, but environment entries are meant to substitute t for x on some other term, while slices are meant to receive a variable and substitute it for x in t.

The Sliced POM: New Transition. The Sliced POM inherits the same four transitions of the Natural POM, but note that in transition $\rightsquigarrow_{\mathsf{m}}$ now it is no longer necessary to decompose the body u of the abstraction as $O\langle z\rangle$ (please note that u is a meta-variable for *terms*, and not for variables). Additionally, the Sliced POM has a new search transition $\mathtt{sea}_3$. When the evaluation of the active slice is over, which happens when one is left only with the head variable z of the slice, the new transitions $\mathtt{sea}_3$ pops the first slice on the slice stack and replaces z for x in t. An invariant shall guarantee that t is a sub-term of the initial term, so that the cost of the renaming now is under control.

A Technical Point. The attentive reader might wonder why $\rightsquigarrow_{\mathsf{m}}$ and $\rightsquigarrow_{\mathtt{sea}_3}$ are not reformulated in the following eager way (with slices reduced to be simply terms), where the head of the abstraction is substituted when the slice is *pushed* on the slice stack, and not when it is popped:

Sl. st.	Active slice	Env		Sl. st.	Active slice	Env
$\mathtt{S}$	$t[x\leftarrow(\lambda y.O\langle z\rangle)w]$	$\mathtt{E}$	$\rightsquigarrow_{\mathsf{m}}$	$\mathtt{S}:t\{x\leftarrow z\}$	$O\langle z\rangle\{y\leftarrow w\}$	$\mathtt{E}$
$\mathtt{S}:t$	z	$\mathtt{E}$	$\rightsquigarrow_{\mathtt{sea}_3}$	$\mathtt{S}$	t	$\mathtt{E}$

We do not adopt the eager approach because the evaluation of the active slice might change its head variable (thus making it unsound to substitute eagerly), as it is demonstrated by the following run of the standard Sliced POM:

Slice stack	Active slice	Env	
ϵ	$t[x\leftarrow(\lambda y.z[z\leftarrow(\lambda x'.x')w'])w]$	ϵ	$\rightsquigarrow_{\mathsf{m}}$
$\epsilon:t[x\leftarrow\cdot]$	$z[z\leftarrow(\lambda x'.x')w']$	ϵ	$\rightsquigarrow_{\mathsf{m}}$
$\epsilon:t[x\leftarrow\cdot]:z[z\leftarrow\cdot]$	w'	ϵ	$\rightsquigarrow_{\mathtt{sea}_3}$
$\epsilon:t[x\leftarrow\cdot]$	w'	ϵ	

Invariants. To establish the bisimulation between the strategy $\rightarrow_{\mathtt{r}}$ of the positive λ-calculus and the Sliced POM, we shall prove that the two form a distillery, that, in turn, is proved using the following invariants of the machine.

Lemma 4 (Qualitative invariants). *Let* $\mathtt{Q}=(\mathtt{S},t,\mathtt{E})$ *be a reachable state.*

1. Contextual read-back*:* $\overline{\mathtt{E}}$ *is a right context.*
2. Well-bound*:*

– Bound names in terms*: if $\lambda x.u$ or $u[x\leftarrow b]$ occurs in $\mathtt{Q}$ then any other occurrence of x in $\mathtt{Q}$, if any, is a free variable occurrence of u;*
– ES names of environment entries*: for any ES $[y\leftarrow b]$ in $\mathtt{E}$ the name y can occur (in any form) only on the left of that ES in $\mathtt{Q}$.*

Proof. By induction on the length of the run reaching $\mathtt{Q}$. For both points, the base case trivially holds, and the inductive case is by analysis of the last transition, which is always a straightforward inspection of the transitions using the *i.h.* For contextual read-back, the proof relies on the inside-out definition of right contexts (see Def. 2). □

The well-bound invariant has two consequences that might not be evident. Firstly, there cannot be two ESs $[x\leftarrow b]$ and $[x\leftarrow b']$ on the same variable. This fact implies the determinism of the machine, that however shall not be proved because it is not necessary (determinism of the strategy is enough for proving the bisimulation). Secondly, the name x of an environment entry $[x\leftarrow b]$ can occur both in the active slice and in the slice stack, while the names of ESs in slices can occur only within the slice. In particular, x cannot occur in $\mathtt{S}$ in $(\mathtt{S}, t[x\leftarrow b], \mathtt{E})$.

Theorem 3 (Distillery).

1. *Principal projection:*
   (a) *if* $\mathtt{Q} \rightsquigarrow_{\mathsf{e}} \mathtt{Q}'$ *then* $\overline{\mathtt{Q}} \to_{\mathsf{e}} \overline{\mathtt{Q}'}$.
   (b) *if* $\mathtt{Q} \rightsquigarrow_{\mathsf{m}} \mathtt{Q}'$ *then* $\overline{\mathtt{Q}} \to_{\mathsf{m}} \overline{\mathtt{Q}'}$.
2. *Search transparency: if* $\mathtt{Q} \rightsquigarrow_{\mathtt{sea}_1,\mathtt{sea}_2,\mathtt{sea}_3} \mathtt{Q}'$ *then* $\overline{\mathtt{Q}} = \overline{\mathtt{Q}'}$.
3. *Search terminates:* $\rightsquigarrow_{\mathtt{sea}_1,\mathtt{sea}_2,\mathtt{sea}_3}$ *is terminating.*
4. *Halt: if* $\mathtt{Q}$ *is final then it is of the form* $(\epsilon, x, \mathtt{E})$ *and* $\overline{\mathtt{Q}} = \overline{\mathtt{E}}\langle x\rangle$ *is* $\to_{\mathtt{pos}}$*-normal.*

The points of the proved theorem together with determinism of the right strategy (Lemma 3) provide all the requirements for a distillery. Then, the abstract theorem about distilleries (Thm. 2) gives the following corollary.

Corollary 1 (Mechanical bisimulation). *The Sliced POM and the right strategy* $\to_{\mathtt{r}}$ *are in a mechanical bisimulation.*

7 Complexity Analysis

In this section, we show that the Sliced POM can be concretely implemented within a bi-linear overhead, that is, linear in the number of m-steps/transitions and the size of the initial term.

Sub-Term Property. As it is standard, the complexity analysis crucially relies on the sub-term property. In CbV settings, the property is usually expressed saying that all values are sub-terms of the initial terms, as in Lemma 1. The Sliced POM only duplicates values too, but (as we have discussed for the Natural POM) we also want to know that the terms to which renamings are applied are sub-terms

of the initial term, and these terms are not values. Thus, the property is given with respect to *all* terms in a state.

There is in fact a very minor exception. The substitution performed by an exponential transition takes two sub-terms of the initial term, namely $t[x{\leftarrow}yz]$ and $\lambda w.u$, and creates a term $t[x{\leftarrow}(\lambda w.u)^{\alpha}z]$ that is not a sub-term of the initial one. The new term, however, is very short lived: the next transition is multiplicative and decomposes that term in sub-terms of the initial term (up to renaming).

Lemma 5 (Sub-term property). *Let $t \triangleleft \mathtt{Q} \rightsquigarrow^* \mathtt{Q}' = (\mathtt{S}, u, \mathtt{E})$ be a Sliced POM run. Then:*

1. *If $\mathtt{Q}'$ is not the target of a e-transition then $|r| \leq |t|$ for any term r in $\mathtt{Q}'$;*
2. *Otherwise, $|r| \leq |t|$ for any term r in $\mathtt{Q}'$ except u.*

Proof. By induction on the length of the run, inspecting the last transition and using the *i.h.* □

Number of Transitions. Some basic observations about the transitions, together with the sub-term property, allow us to bound their number using the two key parameters (that is, number of m-steps/transitions and size of the initial term).

Lemma 6 (Number of transitions). *Let $r : t \triangleleft \mathtt{Q} \rightsquigarrow^* \mathtt{Q}'$ be a Sliced POM run.*

1. *$|r|_{\mathsf{e},\mathtt{sea}_3} \in \mathcal{O}(|r|_{\mathsf{m}})$;*
2. *$|r|_{\mathtt{sea}_1,\mathtt{sea}_2} \in \mathcal{O}(|t| \cdot (|r|_{\mathsf{m}} + 1))$.*

Proof. 1. $|r|_{\mathsf{e}} \leq |r|_{\mathsf{m}} + 1$ because exponential transitions can only be followed by multiplicative transitions.
$|r|_{\mathtt{sea}_3} \leq |r|_{\mathsf{m}}$ because every $\mathtt{sea}_3$ transition consumes one entry from the slice stack, which are created only by m transitions.

2. Note that $\mathtt{sea}_1/\mathtt{sea}_2$ transitions decrease the size of the active slice, which is increased only by transitions e and $\mathtt{sea}_3$. By the sub-term property (Lemma 5), the size increase of the active slice by transitions e and $\mathtt{sea}_3$ is bounded by the size $|t|$ of the initial term. By Point 1, $|r|_{\mathsf{e},\mathtt{sea}_3} = \mathcal{O}(|r|_{\mathsf{m}})$. Then $|r|_{\mathtt{sea}_1,\mathtt{sea}_2} \in \mathcal{O}(|t| \cdot (|r|_{\mathsf{e},\mathtt{sea}_2} + 1)) = \mathcal{O}(|t| \cdot (|r|_{\mathsf{m}} + 1))$. □

Cost of Single Transitions. Lastly, we need some assumptions on how the Sliced POM can be concretely implemented. Transition $\mathtt{sea}_2$ can evidently be done in $\mathcal{O}(1)$. Our OCaml implementation—overviewed in Appendix A of the technical report [18]—represents variables as memory locations and variable occurrences as pointers to those locations, obtaining random access to environment entries in $\mathcal{O}(1)$.[5] Therefore, also transition $\mathtt{sea}_1$ can be done in $\mathcal{O}(1)$. The cost of transition

[5] Assuming that memory accesses take $\mathcal{O}(1)$ is an idealized abstraction. In real computer architectures, that cost is highly dependent on where the data is stored, but it is bounded nonetheless. To be theoretically precise, the cost of accessing an *unbound* memory should be instead assumed to be logarithmic in the size of the memory in use. In the analyses of polynomial algorithms, and of abstract machines in particular, it is standard to assume an underlying random access machine model with constant-time access, since the logarithmic factor is somewhat negligible. The index of the location does instead play a more relevant role in the study of lower time and space complexities.

$\mathtt{e}$ is bound by the size of the value to copy, itself bound by the sub-term property. The cost of transitions $\mathtt{m}$ and $\mathtt{sea}_3$ is bound by the size of the term to rename, itself bound by the sub-term property. The next lemma sums it up.

Lemma 7 (Cost of single transitions). *Let $t \triangleleft \mathtt{Q} \rightsquigarrow^* \mathtt{Q}'$ be a Sliced POM run. Implementing $\mathtt{sea}_1$ and $\mathtt{sea}_2$ transitions from $\mathtt{Q}'$ costs $\mathcal{O}(1)$ each while implementing $\mathtt{e}$, $\mathtt{m}$, and $\mathtt{sea}_3$ transitions costs $\mathcal{O}(|t|)$ each.*

Putting all together, we obtain our main result: a bilinear bound for the Sliced POM, showing that it is an efficient machine for the right strategy.

Theorem 4 (Sliced POM is bi-linear). *Let $r : t \triangleleft \mathtt{Q} \rightsquigarrow^* \mathtt{Q}'$ be a Sliced POM run. Then r can be implemented on random access machines in $\mathcal{O}(|t| \cdot (|r|_{\mathtt{m}} + 1))$.*

Proof. The cost of implementing r is obtained by multiplying the number of each kind of transitions (Lemma 6) by the cost of that kind of transition (Lemma 7), and summing over all kinds of transition. □

8 Conclusions

In λ-calculi with sharing, renaming chains are a recurrent issue that causes both time and space inefficiencies. The recently introduced positive λ-calculus removes renaming chains, while adding some meta-level renamings. This paper stems from the observation that the added meta-level renamings reintroduce a time inefficiency if implemented naively.

The problem is analyzed via the sub-term property, showing that the culprit is the fact that the scope of renamings is not a sub-term of the initial term. The analysis leads to the design of an optimized machine, the new Sliced POM, that removes once and for all the inefficiency. The key tool is a new decomposition in slices of the scopes of renamings, via a new stack for slices playing a role dual to that of environments. We also provide a prototype OCaml implementation of the Sliced POM, described in Appendix A of the technical report [18].

Future Work. At the theoretical level, we plan to adapt the positive λ-calculus and the Sliced POM to call-by-need evaluation. At the practical level, it would be interesting to see how the schema of the Sliced POM combines with other techniques such as closure conversion or skeletal call-by-need; these techniques were in fact recasted in the same abstract machine framework of the present work, and also analyzed from a complexity point of view, in two parallel works involving Accattoli and Sacerdoti Coen [11,15]. We also suspect that the Sliced POM can be used to simplify the sophisticated machine for Strong Call-by-Value by Accattoli et al. in [8].

Acknowledgments. The second author is funded by the INdAM/GNCS project MARQ and the third by the ANR project RECIPROG (ANR-21-CE48-019).

References

1. Abadi, M., Cardelli, L., Curien, P., Lévy, J.: Explicit substitutions. J. Funct. Program. **1**(4), 375–416 (1991). https://doi.org/10.1017/S0956796800000186
2. Accattoli, B.: Exponentials as substitutions and the cost of cut elimination in linear logic. Log. Methods Comput. Sci. **19**(4) (2023). https://doi.org/10.46298/LMCS-19(4:23)2023
3. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling abstract machines. In: 19th ACM SIGPLAN International Conference on Functional Programming, ICFP 2014. pp. 363–376. ACM (2014). https://doi.org/10.1145/2628136.2628154
4. Accattoli, B., Barenbaum, P., Mazza, D.: A strong distillery. In: Feng, X., Park, S. (eds.) Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9458, pp. 231–250. Springer (2015). https://doi.org/10.1007/978-3-319-26529-2_13
5. Accattoli, B., Barras, B.: Environments and the complexity of abstract machines. In: Vanhoof, W., Pientka, B. (eds.) Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017. pp. 4–16. ACM (2017). https://doi.org/10.1145/3131851.3131855
6. Accattoli, B., Bonelli, E., Kesner, D., Lombardi, C.: A nonstandard standardization theorem. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 659–670. ACM (2014). https://doi.org/10.1145/2535838.2535886
7. Accattoli, B., Condoluci, A., Guerrieri, G., Sacerdoti Coen, C.: Crumbling abstract machines. In: Komendantskaya, E. (ed.) Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019. pp. 4:1–4:15. ACM (2019). https://doi.org/10.1145/3354166.3354169
8. Accattoli, B., Condoluci, A., Sacerdoti Coen, C.: Strong call-by-value is reasonable, implosively. In: 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021. pp. 1–14. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470630
9. Accattoli, B., Dal Lago, U.: (leftmost-outermost) beta reduction is invariant, indeed. Log. Methods Comput. Sci. **12**(1) (2016). https://doi.org/10.2168/LMCS-12(1:4)2016
10. Accattoli, B., Dal Lago, U., Vanoni, G.: Reasonable space for the λ-calculus, logarithmically. In: Baier, C., Fisman, D. (eds.) LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022. pp. 47:1–47:13. ACM (2022). https://doi.org/10.1145/3531130.3533362
11. Accattoli, B., Ghica, D., Guerrieri, G., Lourenço, C.B., Sacerdoti Coen, C.: Closure conversion, flat environments, and the complexity of abstract machines. CoRR **abs/2507.15843** (2025). https://doi.org/10.48550/ARXIV.2507.15843, accepted at PPDP 2025
12. Accattoli, B., Guerrieri, G.: Open call-by-value. In: Igarashi, A. (ed.) Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10017, pp. 206–226 (2016). https://doi.org/10.1007/978-3-319-47958-3_12

13. Accattoli, B., Guerrieri, G.: Abstract machines for open call-by-value. Sci. Comput. Program. **184** (2019). https://doi.org/10.1016/J.SCICO.2019.03.002
14. Accattoli, B., Kesner, D.: The structural *lambda*-calculus. In: Dawar, A., Veith, H. (eds.) Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6247, pp. 381–395. Springer (2010). https://doi.org/10.1007/978-3-642-15205-4_30
15. Accattoli, B., Magliocca, F., Peyrot, L., Sacerdoti Coen, C.: The cost of skeletal call-by-need, smoothly. In: Fernández, M. (ed.) 10th International Conference on Formal Structures for Computation and Deduction, FSCD 2025, July 14-20, 2025, Birmingham, UK. LIPIcs, vol. 337, pp. 5:1–5:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025). https://doi.org/10.4230/LIPICS.FSCD.2025.5
16. Accattoli, B., Sacerdoti Coen, C.: On the relative usefulness of fireballs. In: 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015. pp. 141–155. IEEE Computer Society (2015). https://doi.org/10.1109/LICS.2015.23
17. Accattoli, B., Sacerdoti Coen, C.: On the value of variables. Information and Computation **255**, 224–242 (2017). https://doi.org/10.1016/j.ic.2017.01.003
18. Accattoli, B., Sacerdoti Coen, C., Wu, J.: Positive sharing and abstract machines. CoRR **abs/2506.14131** (2025). https://doi.org/10.48550/ARXIV.2506.14131
19. Accattoli, B., Wu, J.H.: Positive focusing is directly useful. Electronic Notes in Theoretical Informatics and Computer Science **Volume 4 - Proceedings of MFPS XL**, 3 (Dec 2024). https://doi.org/10.46298/entics.14758
20. Dal Lago, U., Martini, S.: The weak lambda calculus as a reasonable machine. Theor. Comput. Sci. **398**(1-3), 32–50 (2008). https://doi.org/10.1016/J.TCS.2008.01.044
21. Danos, V., Herbelin, H., Regnier, L.: Game semantics & abstract machines. In: Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996. pp. 394–405. IEEE Computer Society (1996). https://doi.org/10.1109/LICS.1996.561456
22. Felleisen, M., Friedman, D.P.: Control operators, the secd-machine, and the λ-calculus. In: Wirsing, M. (ed.) Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986. pp. 193–222. North-Holland (1987)
23. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Cartwright, R. (ed.) Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993. pp. 237–247. ACM (1993). https://doi.org/10.1145/155090.155113
24. Friedman, D.P., Ghuloum, A., Siek, J.G., Winebarger, O.L.: Improving the lazy krivine machine. High. Order Symb. Comput. **20**(3), 271–293 (2007). https://doi.org/10.1007/S10990-007-9014-0
25. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: Wand, M., Jones, S.L.P. (eds.) Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002. pp. 235–246. ACM (2002). https://doi.org/10.1145/581478.581501
26. Launchbury, J.: A natural semantics for lazy evaluation. In: Deusen, M.S.V., Lang, B. (eds.) Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT

Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993. pp. 144–154. ACM Press (1993). https://doi.org/10.1145/158511.158618
27. Levy, P.B., Power, J., Thielecke, H.: Modelling environments in call-by-value programming languages. Inf. Comput. **185**(2), 182–210 (2003). https://doi.org/10.1016/S0890-5401(03)00088-9
28. Miller, D., Wu, J.H.: A positive perspective on term representation (invited talk). In: Klin, B., Pimentel, E. (eds.) 31st EACSL Annual Conference on Computer Science Logic, CSL 2023, February 13-16, 2023, Warsaw, Poland. LIPIcs, vol. 252, pp. 3:1–3:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.CSL.2023.3
29. Moggi, E.: Computational λ-Calculus and Monads. LFCS report ECS-LFCS-88-66, University of Edinburgh (1988), http://www.lfcs.inf.ed.ac.uk/reports/88/ECS-LFCS-88-66/ECS-LFCS-88-66.pdf
30. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989. pp. 14–23. IEEE Computer Society (1989). https://doi.org/10.1109/LICS.1989.39155
31. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. In: White, J.L. (ed.) Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992. pp. 288–298. ACM (1992). https://doi.org/10.1145/141471.141563
32. Sands, D., Gustavsson, J., Moran, A.: Lambda Calculi and Linear Speedups. In: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. pp. 60–84 (2002). https://doi.org/10.1007/3-540-36377-7_4
33. Sestoft, P.: Deriving a lazy abstract machine. J. Funct. Program. **7**(3), 231–264 (1997). https://doi.org/10.1017/S0956796897002712
34. Wand, M.: On the correctness of the krivine machine. High. Order Symb. Comput. **20**(3), 231–235 (2007). https://doi.org/10.1007/S10990-007-9019-8
35. Wu, J.H.: Proofs as terms, terms as graphs. In: Hur, C. (ed.) Programming Languages and Systems - 21st Asian Symposium, APLAS 2023, Taipei, Taiwan, November 26-29, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14405, pp. 91–111. Springer (2023). https://doi.org/10.1007/978-981-99-8311-7_5