A Strong Distillery

Beniamino Accattoli, Pablo Barenbaum, Damiano Mazza

A Strong Distillery

Beniamino Accattoli[1], Pablo Barenbaum[2], and Damiano Mazza[3]

[1] INRIA & LIX, École Polytechnique
beniamino.accattoli@inria.fr
[2] University of Buenos Aires – CONICET
pbarenbaum@dc.uba.ar
[3] CNRS, UMR 7030, LIPN, Université Paris 13, Sorbonne Paris Cité
Damiano.Mazza@lipn.univ-paris13.fr

Abstract. Abstract machines for the strong evaluation of λ-terms (that is, under abstractions) are a mostly neglected topic, despite their use in the implementation of proof assistants and higher-order logic programming languages. This paper introduces a machine for the simplest form of strong evaluation, leftmost-outermost (call-by-name) evaluation to normal form, proving it correct, complete, and bounding its overhead. Such a machine, deemed *Strong Milner Abstract Machine*, is a variant of the KAM computing normal forms and using just one global environment. Its properties are studied via a special form of decoding, called a *distillation*, into the Linear Substitution Calculus, neatly reformulating the machine as a standard micro-step strategy for explicit substitutions, namely *linear leftmost-outermost reduction*, *i.e.* the extension to normal form of linear head reduction. Additionally, the overhead of the machine is shown to be linear both in the number of steps and in the size of the initial term, validating its design. The study highlights two distinguished features of strong machines, namely backtracking phases and their interactions with abstractions and environments.

1 Introduction

The computational model behind functional programming is the weak λ-calculus, where *weakness* is the fact that evaluation stops as soon as an abstraction is obtained. Evaluation is usually defined in a small-step way, specifying a strategy for the selection of weak β-redexes. Both the advantage and the drawback of λ-calculus is the lack of a machine in the definition of the model. Unsurprisingly implementations of functional languages have been explored for decades.

Implementation schemes are called *abstract machines*, and usually account for two tasks. First, they switch from small-step to *micro-step* evaluation, delaying the costly meta-level substitution used in small-step operational semantics and replacing it with substitutions of one occurrence at a time, when required. Second, they also *search the next redex* to reduce, walking through the program according to some evaluation strategy. Abstract machines are *machines* because they are deterministic and the complexity of their steps can easily be measured,

and are *abstract* because they omit many details of a real implementation, like the actual representation of terms and data-structures or the garbage collector.

Historically, the theory of λ-calculus and the implementation of functional languages have followed orthogonal approaches. The former rather dealt with *strong* evaluation, and it is only since the seminal work of Abramsky and Ong [1] that the theory took weak evaluation seriously. Dually, practical studies mostly ignored *strong* evaluation, with the notable exception of Crégut [13, 14] (1990) and, more recently, the semi-strong approach of Grégoire and Leroy [23] (2002)—see also the *related work* paragraph below. Strong evaluation is nonetheless essential in the implementation of proof assistants or higher-order logic programming, typically for type-checking in frameworks with dependent types as the Edinburgh Logical Framework or the Calculus of Constructions, as well as for unification modulo $\beta\eta$ in simply typed frameworks like λ-prolog.

The aim of this paper is to move the first steps towards a systematic and theoretical exploration of the implementation of strong evaluation. Here we deal with the simplest possible case, call-by-name evaluation to strong normal form, implemented by a variant of the Krivine Abstract Machine. The study is carried out according to the *distillation methodology*, a new approach recently introduced by the authors and previously applied only to weak evaluation [3].

Distilling Abstract Machines. Many abstract machines can be rephrased as strategies in *λ-calculi with explicit substitutions* (ES for short), see at least [15, 24, 14, 10, 25, 9]. The Linear Substitution Calculus (LSC)—a variation over a λ-calculus with ES by Robin Milner [27] developed by Accattoli and Kesner [2, 5]—provides more than a simple reformulation: it disentangles the two tasks carried out by abstract machines, retaining the *micro-step operational semantics* and omitting the *search for the next redex*. Such a neat disentangling, that we prefer to call a *distillation*, is a decoding based on the following key points:

1. *Partitioning*: the machine transitions are split in two classes. *Principal transitions* are mapped to the rewriting rules of the calculus, while *commutative transitions*—responsible for the search for the redex—are mapped on a notion of structural equivalence, specific to the LSC.
2. *Rewriting*: structural equivalence accounts both for the search for the redex and garbage collection, and commutes with evaluation. It can thus be postponed, isolating the micro-step strategy in the rewriting of the LSC.
3. *Logic*: the LSC itself has only two rules, corresponding to cut-elimination in linear logic proof nets. A distillation then provides a logical reading of an abstract machine (see [3] for more details).
4. *Complexity*: by design, a principal transition has to take linear time in the input, while a commutative transition has to be constant.

A *distillery* is then given by a machine, a strategy, a structural equivalence, and a decoding function satisfying the above points. In bilinear distilleries, the number of commutative transitions is linear in both the *number of principal transitions* and the *size of the initial term*. Bilinearity guarantees that distilling away the commutative part by switching to the LSC preserves the asymptotical

behavior, *i.e.* it does not forget too much. At the same time, the bound on the commutative overhead justifies the design of the abstract machine, providing a provably bounded implementation scheme.

A Strong Distillery. Our machine is a strong version of the Milner Abstract Machine (MAM), a variant with just one *global environment* of the Krivine Abstract Machine (KAM), introduced in [3].

The first result of the paper is the design of a distillery relating the Strong MAM to *linear leftmost-outermost reduction* in the LSC [5, 6]—that is at the same time a refinement of leftmost-outermost (LO) β-reduction and an extension of linear head reduction [26, 16, 2] to normal form—together with the proof of correctness and completeness of the implementation. Moreover, the linear LO strategy is *standard* and *normalizing* [5], and thus we provide an instance of Plotkin's approach of mapping abstract machines to such strategies [28].

The second result is the complexity analysis showing that the distillery is bilinear, *i.e.* that the cost of the additional search for the next redex specific to the machine is negligible. The analysis is simple, and yet subtle and robust. It is subtle because it requires a global analysis of executions, and it is robust because the overhead is bilinear for *any* evaluation sequence, not necessarily to normal form, and even for diverging ones.

For the design of the Strong MAM we make various choices:

1. *Global Environment*: we employ a *global* environment, which is in opposition to having closures (pairing subterms with *local* environments), and it models a store-based implementation scheme. The choice is motivated by future extensions to more efficient strategies as call-by-need, where the global environment allows to integrate sharing with a form of memoization [18, 3].
2. *Sequential Exploration and Backtracking*: we fix a sequential exploration of the term (according to the leftmost-outermost order), in opposition to the parallel evaluation of the arguments (once a head normal form has been reached). This choice internalizes the handling of the recursive iterations, that would be otherwise left to the meta-level, providing a finer study of the data-structures needed by a strong machine. On the other hand, it forces to have backtracking transitions, activated when the current subterm has been checked to be normal and evaluation needs to retrieve the next subterm on the stack. Call-by-value machines usually have a similar but simpler backtracking mechanism, realized via an additional component, the *dump*.
3. *(Almost) No Garbage Collection*: we focus on time complexity, and thus ignore space issues, that is, our machine does not account for garbage collection. In particular, we keep the global environment completely unstructured, similarly to the (weak) MAM. Strong evaluation however is subtler, as to establish a precise relationship between the machine and the calculus with ES, garbage collection cannot be completely ignored. Our approach is to isolate it within the meta-level: we use a system of parenthesized markers, to delimit subenvironments created under abstractions that could be garbage collected once the machine backtracks outside those abstraction. These labels are not inspected by the transitions, and play a role only for the proof of

the distillation theorem. Garbage collection then is somewhat accounted for by the analysis, but there are no dedicated transitions nor rewriting rules, it is rather encapsulated in the decoding and in the structural equivalence.

Efficiency? It is known that LO evaluation is not efficient. Improvements are possible along three axis: refining the strategy (by turning to strong call-by-value/need, partially done in [23, 14, 8]), speeding up the substitution process (by forbidding the substitution of variables, see [7, 8]), and avoiding useless substitutions (by adding *useful sharing*, see [6, 8]). These improvements however require sophisticated machines, left to future work.

LO evaluation is nonetheless a good first case study, as it allows to isolate the analysis of backtracking phases and their subtle interactions with abstractions and environments. We expect that the mentioned optimizations can be added in a quite modular way, as they have all been addressed in the complementary study in [8], based on the same technology (*i.e.* LSC and distilleries).

(Scarce) Related Work. Beyond Crégut's [13, 14], we are aware of only two other similar works on strong abstract machines, García-Pérez, Nogueira and Moreno-Navarro's [22] (2013), and Smith's [30] (unpublished, 2014). Two further studies, de Carvalho's [12] and Ehrhard and Regnier's [20], introduce strong versions of the KAM but for theoretical purposes; in particular, their design choices are not tuned towards implementations (*e.g.* rely on a naïve parallel exploration of the term). Semi-strong machines for call-by-value (*i.e.* dealing with weak evaluation but on open terms) are studied by Grégoire and Leroy [23] and in a recent work by Accattoli and Sacerdoti Coen [8] (see [8] for a comparison with [23]). More recent work by Dénès [19] and Boutiller [11] appeared in the context of term evaluation in Coq. These works, which do offer the nice perspective of concretely dealing with proof assistants, are focused on quite specific Coq-related tasks (such as term simplification) and the difference in reduction strategy and underlying motivations makes a comparison difficult.

Of all the above, the closest to ours is Crégut's work, because it defines an implementation-oriented strong KAM, thus also addressing leftmost-outermost reduction. His machine uses local environments, sequential exploration and backtracking, scope markers akin to ours, and a calculus with ES to establish the correctness of the implementation. His calculus, however, has no less than 13 rewriting rules, while ours just 2, and so our approach is simpler by an order of magnitude. Moreover, we want to stress that our contribution does not lie in the machine *per se*, or the chosen reduction strategy (as long as it is strong), but in the combined presence of a robust and simple abstraction of the machine, provided by the LSC, and the complexity analysis showing that such an abstraction does not miss too much. In this respect, none of the above works comes with an analysis of the overhead of the machine nor with the logical and rewriting perspective we provide. In fact, our approach offers general guidelines for the design of (strong) abstract machines. The choice of leftmost-outermost reduction showcases the idea while keeping technicalities to a minimum, but it is by no means a limitation. The development of strong distilleries for call-by-value

or lazy strategies, which may be more attractive from a programming languages perspective, are certainly possible and will be the object of future work (again, an intermediary step has already been taken in [8]).

Global environments are explored by Fernández and Siafakas in [21], and used in a minority of works, *e.g.* [29, 18]. We introduced the distillation technique in [3] to revisit the relationship between the KAM and weak linear head reduction pointed out by Danos and Regnier [16]. Distilleries have also been used in [8]. The idea to distinguish between *operational content* and *search for the redex* in an abstract machine is not new, as it underlies in particular the *refocusing semantics* of Danvy and Nielsen [17]. The LSC, with its roots in linear logic proof nets, allows to see this distinction as an avatar of the principal/commutative divide in cut-elimination, because machine transitions may be seen as cut-elimination steps [9, 3]. Hence, it is fair to say that distilleries bring an original refinement where logic, rewriting, and complexity enlighten the picture, leading to formal bounds on machine overheads.

Omitted proofs may be found in [4].

2 Linear Leftmost-Outermost Reduction

The language of the *linear substitution calculus* (LSC for short) is given by the following term grammar:

$$\text{LSC Terms} \qquad t, u, w, r ::= x \mid \lambda x.t \mid tu \mid t[x{\leftarrow}u].$$

The constructor $t[x{\leftarrow}u]$ is called an *explicit substitution, shortened ES* (of u for x in t). Both $\lambda x.t$ and $t[x{\leftarrow}u]$ bind x in t, and we silently work modulo α-equivalence of these bound variables, *e.g.* $(xy)[y{\leftarrow}t]\{x{\leftarrow}y\} = (yz)[z{\leftarrow}t]$.

The operational semantics of the LSC is parametric in a notion of (one-hole) context. General *contexts*, that simply extend the contexts for λ-terms with the two cases for ES, and the special case of *substitution contexts* are defined by:

$$\begin{array}{rl} \text{Contexts} & C, C' ::= \langle\cdot\rangle \mid \lambda x.C \mid Ct \mid tC \mid C[x{\leftarrow}t] \mid t[x{\leftarrow}C]; \\ \text{Substitution Contexts} & L, L' ::= \langle\cdot\rangle \mid L[x{\leftarrow}t]. \end{array}$$

The *plugging* $C\langle t\rangle$ of a term t into a context C is defined as $\langle\cdot\rangle\langle t\rangle := t$, $(\lambda x.C)\langle t\rangle := \lambda x.(C\langle t\rangle)$, and so on. As usual, plugging in a context can capture variables, *e.g.* $((\langle\cdot\rangle y)[y{\leftarrow}t])\langle y\rangle = (yy)[y{\leftarrow}t]$. The plugging $C\langle C'\rangle$ of a context C' into a context C is defined analogously.

We write $C \prec_p t$ if there is a term u s.t. $C\langle u\rangle = t$, call it the *prefix relation*.

The rewriting relation is $\to := \to_{\mathtt{m}} \cup \to_{\mathtt{e}}$ where $\to_{\mathtt{m}}$ and $\to_{\mathtt{e}}$ are the *multiplicative* and *exponential* rules, defined by

	Rule at Top Level	Contextual closure	
Multiplicative	$L\langle\lambda x.t\rangle u \mapsto_{\mathtt{m}} L\langle t[x{\leftarrow}u]\rangle$	$C\langle t\rangle \to_{\mathtt{m}} C\langle u\rangle$	if $t \mapsto_{\mathtt{m}} u$
Exponential	$C\langle x\rangle[x{\leftarrow}u] \mapsto_{\mathtt{e}} C\langle u\rangle[x{\leftarrow}u]$	$C\langle t\rangle \to_{\mathtt{e}} C\langle u\rangle$	if $t \mapsto_{\mathtt{e}} u$

The rewriting rules are assumed to use *on-the-fly* α-equivalence to avoid variable capture. For instance, $(\lambda x.t)[y{\leftarrow}u]y \to_{\mathtt{m}} t\{y{\leftarrow}z\}[x{\leftarrow}y][z{\leftarrow}u]$ for $z \notin \mathtt{fv}(t)$,

and $(\lambda y.(xy))[x{\leftarrow}y] \to_{\mathtt{e}} (\lambda z.(yz))[x{\leftarrow}y]$. Moreover, in $\to_{\mathtt{e}}$ the context C is assumed to not capture x, in order to have $(\lambda x.x)[x{\leftarrow}y] \not\to_{\mathtt{e}} (\lambda x.y)[x{\leftarrow}y]$.

The above operational semantics ignores garbage collection. In the LSC, this may be realized by an additional rule which may always be postponed, see [2].

Taking the external context into account, an exponential step has the form $C'\langle C\langle x\rangle[x{\leftarrow}u]\rangle \to_{\mathtt{e}} C'\langle C\langle u\rangle[x{\leftarrow}u]\rangle$. We shall often use a *compact* form:

Exponential Rule in Compact Form
$$C''\langle x\rangle \to_{\mathtt{e}} C''\langle u\rangle \qquad \text{if } C'' = C'\langle C[x{\leftarrow}u]\rangle$$

Definition 1 (Redex Position). *Given a $\to_{\mathtt{m}}$-step $C\langle t\rangle \to_{\mathtt{m}} C\langle u\rangle$ with $t \mapsto_{\mathtt{m}} u$ or a compact $\to_{\mathtt{e}}$-step $C\langle x\rangle \to_{\mathtt{e}} C\langle t\rangle$, the* position *of the redex is the context C.*

We identify a redex with its position, thus using C, C', C'' for redexes, and use $d : t \to^k u$ for derivations, *i.e.* for possibly empty sequences of rewriting steps. We write $|t|_{[\cdot]}$ for the number of substitutions in t, and use $|d|$, $|d|_{\mathtt{m}}$, and $|d|_{\mathtt{e}}$ for the number of steps, $\mathtt{m}$-steps, and $\mathtt{e}$-steps in d, respectively.

Linear Leftmost-Outermost Reduction, Two Definitions. We give two definitions of linear LO reduction $\to_{\mathtt{LO}}$, a traditional one based on ordering redexes and a new contextual one not mentioning the order, apt to work with LSC and relate it to abstract machines. We start by defining the LO order on contexts.

Definition 2 (LO Order). *The* outside-in order $C \prec_O C'$ *is defined by*

1. Root*: $\langle\cdot\rangle \prec_O C$ for every context $C \neq \langle\cdot\rangle$;*
2. Contextual closure*: if $C \prec_O C'$ then $C''\langle C\rangle \prec_O C''\langle C'\rangle$ for any context C''.*

Note that $\prec_O$ can be seen as the prefix relation $\prec_p$ on contexts. The left-to-right order $C \prec_L C'$ *is defined by*

1. Application*: if $C \prec_p t$ and $C' \prec_p u$ then $Cu \prec_L tC'$;*
2. Substitution*: if $C \prec_p t$ and $C' \prec_p u$ then $C[x{\leftarrow}u] \prec_L t[x{\leftarrow}C']$;*
3. Contextual closure*: if $C \prec_L C'$ then $C''\langle C\rangle \prec_L C''\langle C'\rangle$ for any context C''.*

Last, the left-to-right outside-in order *is defined by $C \prec_{LO} C'$ if $C \prec_O C'$ or $C \prec_L C'$.*

Two examples of the outside-in order are $(\lambda x.\langle\cdot\rangle)t \prec_O (\lambda x.(\langle\cdot\rangle[y{\leftarrow}u]))t$ and $t[x{\leftarrow}\langle\cdot\rangle] \prec_O t[x{\leftarrow}uC]$, and an example of the left-to-right order is $t[x{\leftarrow}C]u \prec_L t[x{\leftarrow}w]\langle\cdot\rangle$. The next immediate lemma guarantees that we defined a total order.

Lemma 1 (Totality of $\prec_{LO}$). *If $C \prec_p t$ and $C' \prec_p t$ then either $C \prec_{LO} C'$ or $C' \prec_{LO} C$ or $C = C'$.*

Remember that we identify redexes with their position context and write $C \prec_{LO} C'$. We can now define linear LO reduction, first considered in [5], where it is proved that it is standard and normalizing, and then in [6], extending linear head reduction [26, 16, 2] to normal form.

Definition 3 (Linear LO Reduction $\to_{\mathtt{LO}}$). *Let t be a term. C is the* leftmost-outermost *(LO for short) redex of t if $C \prec_{LO} C'$ for every other redex C' of t. We write $t \to_{\mathtt{LO}} u$ if a step reduces the LO redex.*

We now define LO contexts and prove that the position of a linear LO step is always a LO context. We need two notions.

Definition 4 (Neutral Term). *A term is* neutral *if it is $\to$-normal and it is not of the form $L\langle\lambda x.t\rangle$.*

Neutral terms are such that their plugging in a context cannot create a multiplicative redex. We also need the notion of left free variable of a context, *i.e.* of a variable occurring free at the left of the hole.

Definition 5 (Left Free Variables). *The set $\mathtt{lfv}(C)$ of* left free variables *of C is defined by:*

$$\begin{array}{rlrl} \mathtt{lfv}(\langle\cdot\rangle) &:= \emptyset & \mathtt{lfv}(tC) &:= \mathtt{fv}(t) \cup \mathtt{lfv}(C) \\ \mathtt{lfv}(\lambda x.C) &:= \mathtt{lfv}(C) \setminus \{x\} & \mathtt{lfv}(C[x{\leftarrow}t]) &:= \mathtt{lfv}(C) \setminus \{x\} \\ \mathtt{lfv}(Ct) &:= \mathtt{lfv}(C) & \mathtt{lfv}(t[x{\leftarrow}C]) &:= (\mathtt{fv}(t) \setminus \{x\}) \cup \mathtt{lfv}(C) \end{array}$$

Definition 6 (LO Contexts). *A context C is* LO *if*

1. Right Application*: whenever $C = C'\langle tC''\rangle$ then t is neutral, and*
2. Left Application*: whenever $C = C'\langle C''t\rangle$ then $C'' \neq L\langle\lambda x.C'''\rangle$.*
3. Substitution*: whenever $C = C'\langle C''[x{\leftarrow}u]\rangle$ then $x \notin \mathtt{lfv}(C'')$.*

Lemma 2 (LO Reduction and LO Contexts). *Let $t \to u$ by reducing a redex C. Then C is a $\to_{\mathtt{LO}}$ step iff C is LO.*

Structural Equivalence. A peculiar trait of the LSC is that the rewriting rules do not propagate ES. Therefore, evaluation is usually stable by structural equivalences moving ES around. In this paper we use the following equivalence, including garbage collection ($\equiv_{\mathtt{gc}}$), that we prove to be a strong bisimulation.

Definition 7 (Structural equivalence). *The* structural equivalence $\equiv$ *is the symmetric, reflexive, transitive, and contextual closure of the following axioms:*

$$\begin{array}{rll} (\lambda x.t)[y{\leftarrow}u] \equiv_{\lambda} \lambda x.t[y{\leftarrow}u] & & \text{if } x \notin \mathtt{fv}(u) \\ (t\,u)[x{\leftarrow}w] \equiv_{@\mathtt{l}} t[x{\leftarrow}w]\,u & & \text{if } x \notin \mathtt{fv}(u) \\ (t\,u)[x{\leftarrow}w] \equiv_{@\mathtt{r}} t\,u[x{\leftarrow}w] & & \text{if } x \notin \mathtt{fv}(t) \\ t[x{\leftarrow}u][y{\leftarrow}w] \equiv_{\mathtt{com}} t[y{\leftarrow}w][x{\leftarrow}u] & & \text{if } y \notin \mathtt{fv}(u) \text{ and } x \notin \mathtt{fv}(w) \\ t[x{\leftarrow}u][y{\leftarrow}w] \equiv_{[\cdot]} t[x{\leftarrow}u[y{\leftarrow}w]] & & \text{if } y \notin \mathtt{fv}(t) \\ t[x{\leftarrow}u] \equiv_{\mathtt{gc}} t & & \text{if } x \notin \mathtt{fv}(t) \\ t[x{\leftarrow}u] \equiv_{\mathtt{dup}} t_{[y]_x}[x{\leftarrow}u][y{\leftarrow}u] & & \end{array}$$

In $\equiv_{\mathtt{dup}}$, $t_{[y]_x}$ denotes a term obtained from t by renaming some (possibly none) occurrences of x as y, with y a fresh variable.

Proposition 1 (Structural Equivalence $\equiv$ is a Strong Bisimulation). *If $t \equiv u \to_{\mathtt{LO}} w$ then exists r s.t. $t \to_{\mathtt{LO}} r \equiv w$ and the steps are either both multiplicative or both exponential.*

3 Distilleries

An abstract machine $\mathtt{M}$ is meant to implement a strategy $\multimap$ via a *distillation*, *i.e.* a decoding function $\underline{\cdot}$. A machine has a state s, given by a *code* $\overline{t}$, *i.e.* a λ-term t without ES and not considered up to α-equivalence, and some data-structures like stacks, dumps, environments, and heaps. The data-structures are used to implement the search for the next $\multimap$-redex and some form of substitution, and they decode to evaluation contexts for $\multimap$. Every state s decodes to a term $\underline{s}$, having the shape $C_s\langle\overline{t}\rangle$, where $\overline{t}$ is the code currently under evaluation and C_s is the evaluation context given by the data-structures.

A machine computes using transitions, whose union is denoted by $\rightsquigarrow$, of two types. The *principal* one, denoted by $\rightsquigarrow_{\mathtt{p}}$, corresponds to the firing of a rule defining $\multimap$, up to structural equivalence $\equiv$. The *commutative* transitions, denoted by $\rightsquigarrow_{\mathtt{c}}$, only rearrange the data structures, and on the calculus are either invisible or mapped to $\equiv$. The terminology reflects a proof-theoretic view, as machine transitions can be seen as cut-elimination steps [9, 3]. The transformation of evaluation contexts is formalized in the LSC as a structural equivalence $\equiv$, which is required to commute with evaluation $\multimap$, *i.e.* to satisfy

$$\begin{array}{ccc} t & \multimap & r \\ \equiv & & \\ u & & \end{array} \quad \Rightarrow \exists q \text{ s.t.} \quad \begin{array}{ccc} t & \multimap & r \\ \equiv & & \equiv \\ u & \multimap & q \end{array}$$

for each of the rules of $\multimap$, preserving the kind of rule. In fact, this means that $\equiv$ is a *strong* bisimulation (*i.e. one* step to *one* step) with respect to $\multimap$, that is what we proved in Proposition 1 for the equivalence at work in this paper. Strong bisimulations formalize transformations which are transparent with respect to the behavior, even at the level of complexity, because they can be delayed without affecting the length of evaluation:

Lemma 3 (Postponement of $\equiv$). *If $\equiv$ is a strong bisimulation, $t\ (\multimap \cup \equiv)^*\ u$ implies $t \multimap^* \equiv u$ and the number and kind of steps of $\multimap$ in the two reduction sequences is exactly the same.*

We can finally introduce distilleries, *i.e.* systems where a strategy $\multimap$ simulates a machine $\mathtt{M}$ up to structural equivalence $\equiv$ via the decoding $\underline{\cdot}$.

Definition 8. *A* distillery $\mathtt{D} = (\mathtt{M}, \multimap, \equiv, \underline{\cdot})$ *is given by:*

1. *An* abstract machine $\mathtt{M}$*, given by*
   (a) *a deterministic labeled transition system (lts) $\rightsquigarrow$ over states s, with labels in $\{\mathtt{m}, \mathtt{e}, \mathtt{c}\}$; the transitions labelled by $\mathtt{m}, \mathtt{e}$ are called* principal*, the others* commutative*;*
   (b) *a distinguished class of states deemed* initial*, in bijection with closed λ-terms; from these, the* reachable *states are obtained by applying $\rightsquigarrow^*$;*
2. *a deterministic* strategy *$\multimap$, i.e., a deterministic lts over the terms of the LSC induced by some strategy on its reduction rules, with labels in $\{\mathtt{m}, \mathtt{e}\}$.*

3. *a* structural equivalence $\equiv$ *on terms which is a strong bisimulation with respect to* $\multimap$*;*
4. *a* decoding function $\underline{\cdot}$ *from states to terms whose graph, when restricted to reachable states, is a weak simulation up to* $\equiv$ *(the commutative transitions are considered as* τ *actions). More explicitly, for all reachable states:*
   - projection of principal transitions*:* $s \rightsquigarrow_{\mathtt{p}} s'$ *implies* $\underline{s} \multimap_{\mathtt{p}} \equiv \underline{s'}$ *for all* $\mathtt{p} \in \{\mathtt{m}, \mathtt{e}\}$*;*
   - distillation of commutative transitions*:* $s \rightsquigarrow_{\mathtt{c}} s'$ *implies* $\underline{s} \equiv \underline{s'}$*.*

The simulation property is a minimum requirement, but a stronger form of relationship is usually desirable. Additional hypotheses are required in order to obtain the converse simulation and provide complexity bounds.

Terminology: an *execution* ρ is a sequence of transitions from an initial state. With $|\rho|$, $|\rho|_p$ and $|\rho|_c$ we denote respectively the length, the number of principal and commutative transitions of ρ, whereas $|t|$ denotes the size of a term t.

Definition 9 (Distillation Qualities). *A distillery is*

- Reflective *when on reachable states:*
  - Termination*:* $\rightsquigarrow_{\mathtt{c}}$ *terminates;*
  - Progress*: if* s *is final then* $\underline{s}$ *is a* $\multimap$*-normal form.*
- Bilinear *when, given an execution* ρ *from an initial term* t*:*
  - Execution Length*: the number of commutative steps* $|\rho|_c$ *is linear in both* $|t|$ *and* $|\rho|_p$*, i.e.* $|\rho|_c \leq c \cdot (1+|\rho|_p) \cdot |t|$ *for some non-zero constant* c *(when* $|\rho|_p = 0$*,* $O(|t|)$ *time is still needed to recognize that* t *is normal).*
  - Commutative*: each commutative transition is implementable in* $O(1)$ *time on a RAM;*
  - Principal*: each principal transition is implementable in* $O(|t|)$ *time on a RAM.*

A reflective distillery is enough to obtain a weak *bi*simulation between the strategy $\multimap$ and the machine $\mathtt{M}$, up to structural equivalence $\equiv$ (again, the weakness is with respect to commutative transitions). With $|\rho|_{\mathtt{m}}$ and $|\rho|_{\mathtt{e}}$ we denote respectively the number of multiplicative and exponential transitions of ρ.

Theorem 1 (Correctness and Completeness). *Let* $\mathtt{D}$ *be a reflective distillery and* s *an initial state.*

1. Simulation up to $\equiv$*: for every execution* $\rho : s \rightsquigarrow^* s'$ *there is a derivation* $d : \underline{s} \multimap^* \equiv \underline{s'}$ *s.t.* $|\rho|_{\mathtt{m}} = |d|_{\mathtt{m}}$ *and* $|\rho|_{\mathtt{e}} = |d|_{\mathtt{e}}$*.*
2. Reverse Simulation up to $\equiv$*: for every derivation* $d : \underline{s} \multimap^* t$ *there is an execution* $\rho : s \rightsquigarrow^* s'$ *s.t.* $t \equiv \underline{s'}$ *and* $|\rho|_{\mathtt{m}} = |d|_{\mathtt{m}}$ *and* $|\rho|_{\mathtt{e}} = |d|_{\mathtt{e}}$*.*

Bilinearity, instead, is crucial for the low-level theorem.

Theorem 2 (Low-Level Implementation Theorem). *Let* $\multimap$ *be a strategy on terms with ES s.t. there exists a bilinear reflective distillery* $\mathtt{D} = (\mathtt{M}, \multimap, \equiv, \underline{\cdot})$*. Then a derivation* $d : t \multimap^* u$ *is implementable on RAM machines in* $O((1+|d|) \cdot |t|)$ *steps, i.e. bilinear in the size* $|t|$ *of the initial term and the length* $|d|$ *of the derivation.*

Proof. Given $d : t \multimap^n u$ by Theorem 1.2 there is an execution $\rho : s \rightsquigarrow^* s'$ s.t. $u \equiv \underline{s'}$ and $|\rho|_p = |d|$. The cost of implementing ρ is the sum of the costs of implementing the commutative and the principal transitions. By bilinearity, $|\rho|_c = O((1 + |\rho|_p) \cdot |t|)$ and so all the commutative transitions in ρ require $O((1 + |\rho|_p) \cdot |t|)$ steps, because a single one takes a constant number of steps. Again by bilinearity, each principal one takes $O(|t|)$, and so all the principal transitions together require $O(|\rho|_p \cdot |t|)$ steps. □

4 Strengthening the MAM

The machine we are about to introduce implements leftmost-outermost reduction and may therefore be seen as a strong version of the Krivine abstract machine (KAM). However, it differs from the KAM in the fundamental point of using global, as opposed to local, environments. It is therefore more appropriate to say that it is a strong version of the machine we introduced in [3], which we called MAM (Milner abstract machine). Let us briefly recall its definition:

Code	Stack	Env		Code	Stack	Env	
$\overline{t}\overline{u}$	π	E	$\rightsquigarrow_{\mathtt{c_1}}$	$\overline{t}$	$\overline{u} : \pi$	E	
$\lambda x.\overline{t}$	$\overline{u} : \pi$	E	$\rightsquigarrow_{\mathtt{m}}$	$\overline{t}$	π	$[x \leftarrow \overline{u}] : E$	
x	π	E	$\rightsquigarrow_{\mathtt{e}}$	$\overline{t}^{\alpha}$	π	E	if $E(x) = \overline{t}$

Note that the stack and the environment of the MAM contain *codes*, not *closures* as in the KAM. A global environment indeed circumvents the complex mutually recursive notions of *local environment* and *closure*, at the price of the explicit α-renaming $\overline{t}^{\alpha}$ which is applied *on the fly* in $\rightsquigarrow_{\mathtt{e}}$. The price however is negligible, at least theoretically, as the asymptotic complexity of the machine is not affected, see [3] (the same can be said of variable names vs de Bruijn indexes/levels).

We know that the MAM performs *weak* head reduction, whose reduction contexts are (informally) of the form $\langle\cdot\rangle\pi$. This justifies the presence of the stack. It is immediate to extend the MAM so that it performs full head reduction, *i.e.*, so that the head redex is reduced even if it is under an abstraction. Since head contexts are of the form $\Lambda.\langle\cdot\rangle\pi$ (with Λ a list of abstractions), we simply add a stack of abstractions Λ and augment the machine with the following transition:

Abs	Code	Stack	Env		Abs	Code	Stack	Env
Λ	$\lambda x.\overline{t}$	ϵ	E	$\rightsquigarrow_{\mathtt{c_2}}$	$x : \Lambda$	$\overline{t}$	ϵ	E

The other transitions do not touch the Λ stack.

LO reduction is nothing but iterated head reduction. LO reduction contexts, which we formally introduced in Definition 6, when restricted to the pure λ-calculus (without ES) are of the form $\Lambda.rC\pi$, where: Λ and π are as above; r, if present, is a neutral term; and C is either $\langle\cdot\rangle$ or, inductively, a LO context. Then LO contexts may be represented by stacks of triples of the form (Λ, r, π), where r is a neutral term. These stacks of triples will be called *dumps*.

The states of the machine for full LO reduction are as above but augmented with a dump and a *phase* φ, indicating whether we are executing head reduction

(▼) or whether we are backtracking to find the starting point of the next iteration (▲). To the above transitions (which do not touch the dump and are always in the ▼ phase), we add the following:

Abs	Code	Stack	Env	Dump	Ph		Abs	Code	Stack	Env	Dump	Ph
Λ	x	π	E	D	▼	$\rightsquigarrow_{c_3}$	Λ	x	π	E	D	▲
											if $E(x) = \bot$	
$x : \Lambda$	$\bar{t}$	ϵ	E	D	▲	$\rightsquigarrow_{c_5}$	Λ	$\lambda x.\bar{t}$	ϵ	E	D	▲
ϵ	$\bar{u}$	ϵ	E	$(\Lambda, \bar{t}, \pi) : D$	▲	$\rightsquigarrow_{c_7}$	Λ	$\bar{t}\bar{u}$	π	E	D	▲
Λ	$\bar{t}$	$\bar{u} : \pi$	E	D	▲	$\rightsquigarrow_{c_6}$	ϵ	$\bar{u}$	ϵ	E	$(\Lambda, \bar{t}, \pi) : D$	▼

where $E(x) = \bot$ means that the variable x is undefined in the environment E.

In the machine we actually use we join the dump and the Λ stack into the *frame* F, to reduce the number of machine components (the analysis will however somewhat reintroduce the distinction). In the sequel, the reader should bear in mind that a state of the Strong MAM introduced below corresponds to a state of the machine just discussed according to the following correspondence:[4]

	Abs	Code	Stack	Env	Dump	Ph
Discussed Machine:	Λ_0	$\bar{t}$	π	E	$(\Lambda_1, \bar{t}_1, \pi_1) : \cdots : (\Lambda_n, \bar{t}_n, \pi_n)$	φ

	Frame	Code	Stack	Env	Ph
Strong MAM:	$\Lambda_0 : (\bar{t}_1, \pi_1) : \Lambda_1 : \cdots : (\bar{t}_n, \pi_n) : \Lambda_n$	$\bar{t}$	π	E	φ

5 The Strong Milner Abstract Machine

The components and the transitions of the Strong MAM are given by the first two boxes in Fig. 1. As above, we use $\bar{t}, \bar{u}, \ldots$ to denote *codes*, *i.e.*, terms not containing ES and *well-named*, by which mean that distinct binders bind distinct variables and that the sets of free and bound variables are disjoint (codes are not considered up to α-equivalence). The Strong MAM has two phases: *evaluation* (▼) and *backtracking* (▲).

Initial states. The *initial states* of the Strong MAM are of the form $\epsilon \mid \bar{t} \mid \epsilon \mid \epsilon \mid$ ▼, where $\bar{t}$ is a closed code called the *initial term*. In the sequel, we abusively say that a state is reachable from a term meaning that it is reachable from the corresponding initial state.

Scope Markers. The two transitions to evaluate and backtrack on abstractions, $\rightsquigarrow_{▼c_2}$ and $\rightsquigarrow_{▲c_4}$, add markers to delimit subenvironments associated to scopes. The marker ▼x is introduced when the machine starts evaluating under an abstraction λx, while ▲x marks the end of such a subenvironment.

Slight Notation Abuse. The data structures we are going to use are defined as lists, using ϵ for the empty list and ":" for *both* the cons and append operation. The overloading of : means that in the case of, *e.g.*, an environment E we have $E : \epsilon = E = \epsilon : E$, and in particular $\epsilon : \epsilon = \epsilon$. Such an abuse will be used thoughout the whole paper.

[4] Modulo the presence of markers of the form ▲x and ▼x in the environment, which are needed for bookkeeping purposes and were omitted here.

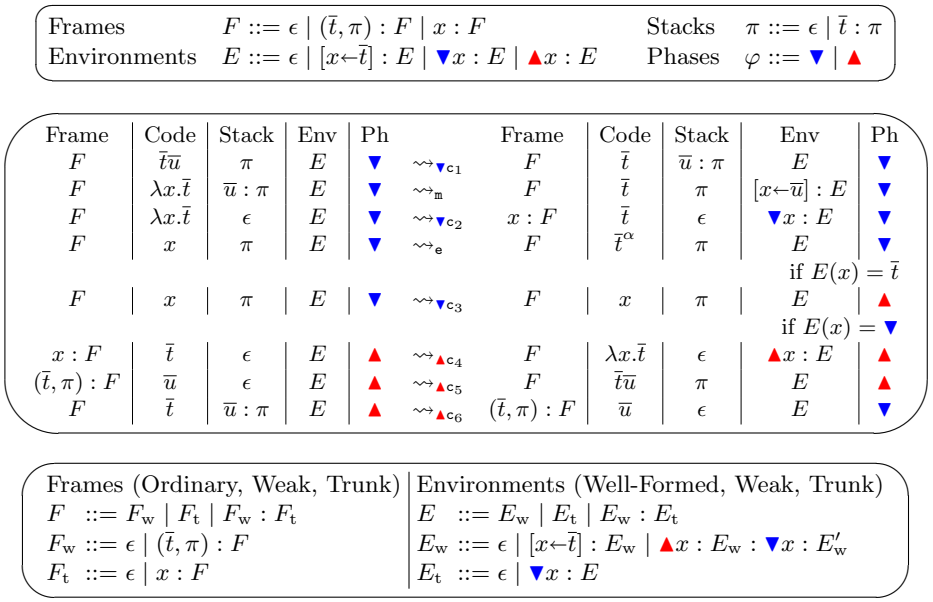

Fig. 1. The Strong MAM.

Weak and Trunk Frames. A frame F may be uniquely decomposed into $F = F_w : F_t$, where $F_w = (\bar{t}_1, \pi_1) : \cdots : (\bar{t}_n, \pi_n)$ (with n possibly null) is a *weak frame*, *i.e.* where no abstracted variable appear, and F_t is a *trunk frame*, *i.e.* not of the form $(\bar{t}, \pi) : F'$ (it either starts a variable entry or it is empty). More precisely, we rely on the alternative grammar in the third box of Fig. 1. We denote by $\Lambda(F)$ the set of variables in F, *i.e.* the set of x s.t. $F = F' : x : F''$.

Weak, Trunk, and Well-Formed Environments. Similarly to the frame, the environment of a reachable state has a weak/trunk structure. In contrast to frames, however, not every environment can be seen this way, but only the well-formed ones (reachable environments will be shown to be well-formed). A weak environment E_w does not contain any open scope, *i.e.* whenever in E_w there is a scope opener marker ($\blacktriangledown x$) then one can also find the scope closer marker ($\blacktriangle x$), and (globally) the closed scopes of E_w are well-parenthesized. A trunk environment E_t may instead also contain open scopes that have no closing marker in E_t (but not unmatched closing markers $\blacktriangle x$). Formally, weak E_w, trunk E_t, and well-formed environments E (all the environments that we will consider will be well-formed, that is why we note them E) are defined in the third box in Fig. 1.

Closed Scopes and Meta-level Garbage Collection. Fragments of the form $\blacktriangle x : E_w : \blacktriangledown x$ within an environment will essentially be ignored; this is how a simple form of garbage collection is encapsulated at the meta-level in the decoding. In

particular, for a well-formed environment E we define $E(x)$ as:

$$\begin{array}{rlrl} \epsilon(x) &:= \bot & (\blacktriangle y : E_{\mathrm{w}} : \blacktriangledown y : E)(x) &:= E(x) \\ ([x{\leftarrow}\overline{t}] : E)(x) &:= \overline{t} & (\blacktriangledown x : E)(x) &:= \blacktriangledown \\ ([y{\leftarrow}\overline{t}] : E)(x) &:= E(x) & (\blacktriangledown y : E)(x) &:= E(x) \end{array}$$

Note that the only potential source of non-determinism for the Strong MAM is the choice among $\rightsquigarrow_{\mathtt{e}}$ and $\rightsquigarrow_{\blacktriangle \mathtt{c}_4}$ in the variable case. The operation $E(x)$, however, is a function, and so the machine is deterministic.

We write $\Lambda(E)$ to denote the set of variables bound to $\blacktriangledown$ by an environment E, *i.e.* those variables whose scope is not closed with $\blacktriangle$.

Lemma 4 (Weak Environments Contain only Closed Scopes). *If E_{w} is a weak environment then $\Lambda(E_{\mathrm{w}}) = \emptyset$.*

Compatibility. In the Strong MAM, both the frame and the environment record information about the abstractions in which evaluation is currently taking place. Clearly, such information has to be coherent, otherwise the decoding of a state becomes impossible. The following compatibility predicate captures the correlation between the structure of the frame and that of the environment.

Definition 10 (Compatibility $F \propto E$). *Compatibility $F \propto E$ between frames and environments is defined by*

1. Base*: $\epsilon \propto \epsilon$;*
2. Weak Extension*: $(F_{\mathrm{w}} : F_{\mathrm{t}}) \propto (E_{\mathrm{w}} : E_{\mathrm{t}})$ if $F_{\mathrm{t}} \propto E_{\mathrm{t}}$;*
3. Abstraction*: $(x : F) \propto (\blacktriangledown x : E)$ if $F \propto E$;*

Lemma 5 (Properties of Compatibility).

1. Well-Formed Environments*: if F and E are compatible then E is well-formed.*
2. Factorization*: every compatible pair $F \propto E$ can be written as $(F_{\mathrm{w}} : F_{\mathrm{t}}) \propto (E_{\mathrm{w}} : E_{\mathrm{t}})$ with $F_{\mathrm{t}} = x : F'$ iff $E_{\mathrm{t}} = \blacktriangledown x : E'$;*
3. Open Scopes Match*: $\Lambda(F) = \Lambda(E)$.*
4. Compatibility and Weak Structures Commute*: for all F_{w} and E_{w}, $F \propto E$ iff $(F_{\mathrm{w}} : F) \propto (E_{\mathrm{w}} : E)$.*

Invariants. The properties of the machine that are needed to prove its correctness and completeness are given by the following invariants.

Lemma 6 (Strong MAM invariants). *Let $s = F \mid \overline{u} \mid \pi \mid E \mid \varphi$ be a state reachable from an initial term $\overline{t}_0$. Then:*

1. Compatibility: *F and E are compatible, i.e. $F \propto E$.*
2. Normal Form:
   (1) Backtracking Code*: if $\varphi = \blacktriangle$, then $\overline{u}$ is normal, and if π is non-empty, then $\overline{u}$ is neutral;*
   (2) Frame*: if $F = F' : (\overline{w}, \pi') : F''$, then $\overline{w}$ is neutral.*

3. Backtracking Free Variables:
   (1) Backtracking Code: *if* $\varphi = \blacktriangle$ *then* $\mathtt{fv}(\overline{u}) \subseteq \Lambda(F)$;
   (2) Pairs in the Frame: *if* $F = F' : (\overline{w}, \pi') : F''$ *then* $\mathtt{fv}(\overline{w}) \subseteq \Lambda(F'')$.
4. Name:
   (1) Substitutions: *if* $E = E' : [x \leftarrow \overline{t}] : E''$ *then* x *is fresh wrt* $\overline{t}$ *and* E'';
   (2) Markers: *if* $E = E' : \blacktriangledown x : E''$ *and* $F = F' : x : F''$ *then* x *is fresh wrt* E'' *and* F'', *and* $E'(y) = \bot$ *for any free variable* y *in* F'';
   (3) Abstractions: *if* $\lambda x.\overline{t}$ *is a subterm of* F, $\overline{u}$, π, *or* E *then* x *may occur only in* $\overline{t}$ *and in the closed subenvironment* $\blacktriangle x : E_{\mathrm{w}} : \blacktriangledown x$ *of* E, *if it exists.*
5. Closure:
   (1) Environment: *if* $E = E' : [x \leftarrow \overline{t}] : E''$ *then* $E''(y) \neq \bot$ *for all* $y \in \mathtt{fv}(\overline{t})$;
   (2) Code, Stack, and Frame: $E(x) \neq \bot$ *for any free variable in* $\overline{u}$ *and in any code of* π *and* F.

Since the statement of the invariants is rather technical, let us summarize the dependencies (or lack thereof) of the various points and their use in the distillation proof of the next section.

- The compatibility, normal form and backtracking free variables invariants are independent of each other and of the subsequent invariants.
- The name invariant relies on the compatibility invariant only.
- The closure invariant relies on the compatibility, name and backtracking free variable invariants only. It is crucial for the progress property (because in the variable case at least one among $\rightsquigarrow_{\mathtt{e}}$ and $\rightsquigarrow_{\blacktriangle \mathtt{c}_4}$ applies).

The proof of every invariant is by induction on the number of transitions leading to the reachable state. In this respect, the various points of the statement of each invariant (*e.g.* points 5.1 and 5.2) are entangled, in the sense that each point needs to use the induction hypothesis of one of the other points, and thus they cannot be proved separately.

Implementing Environments. Note that substitutions in closed scopes are never used by the machine, because the operation $E(x)$ is defined by ignoring them. Moreover, the name invariant guarantees that if $E(x) = \blacktriangledown x$ then E does not contain a substitution on x. These two facts imply that the scope markers $\blacktriangle x$ and $\blacktriangledown x$ are not really needed in an actual implementation: the test $E(x) = \blacktriangledown x$ in $\rightsquigarrow_{\blacktriangledown \mathtt{c}_3}$ can indeed be replaced—in the variant without markers (also redifining $E(x)$ as simple look-up in E)—by a test of undefinedness. The markers are in fact needed only for the analysis, as they structure the frame and the environment of a reachable state into *weak* and *trunk* parts, allowing a simple decoding towards terms with ES.

Moreover, variables are meant to be implemented as memory locations, so that the environment is simply a store, and the list structure of environments is not necessary either. Such an assumption allows to access the environment in constant time on RAM, and will be essential for the proof of the bilinearity of the distillery (to be defined).

Therefore, the structure of environments—given by the scope markers and the list structure—is an artifice used to define the decoding and develop the analysis, but it is not meant to be part of the actual implementation.

6 Distilling the Strong MAM

The definition of the decoding relies on the notion of compatible pair.

Definition 11 (Decoding). *Let $s = (F, \overline{t}, \pi, E, \varphi)$ be a state s.t. $F \propto E$ is a compatible pair. Then s decodes to a state context C_s and a term $\underline{s}$ as follows:*

Weak Environments:

$$\underline{\epsilon} := \langle\cdot\rangle \qquad \underline{[x{\leftarrow}\overline{u}] : E_{\mathrm{w}}} := \underline{E_{\mathrm{w}}}\langle\langle\cdot\rangle[x{\leftarrow}\overline{u}]\rangle \qquad \underline{\blacktriangle x : E_{\mathrm{w}} : \blacktriangledown x : E'_{\mathrm{w}}} := \underline{E'_{\mathrm{w}}}$$

Compatible Pairs:

$$\underline{\epsilon \propto \epsilon} := \langle\cdot\rangle \qquad \underline{(F_{\mathrm{w}} : F_{\mathrm{t}}) \propto (E_{\mathrm{w}} : E_{\mathrm{t}})} := \underline{F_{\mathrm{t}} \propto E_{\mathrm{t}}}\langle\underline{E_{\mathrm{w}}}\langle\underline{F_{\mathrm{w}}}\rangle\rangle \qquad \underline{(x : F) \propto (\blacktriangledown x : E)} := \underline{F \propto E}\langle\lambda x.\langle\cdot\rangle\rangle$$

Weak Frames:

$$\underline{\epsilon} := \langle\cdot\rangle \qquad \underline{(\overline{u}, \pi) : F_{\mathrm{w}}} := \underline{F_{\mathrm{w}}}\langle\underline{\pi}\langle\overline{u}\langle\cdot\rangle\rangle\rangle$$

Stacks:

$$\underline{\epsilon} := \langle\cdot\rangle \qquad \underline{\overline{u} : \pi} := \underline{\pi}\langle\langle\cdot\rangle\overline{u}\rangle$$

States:

$$C_s := \underline{F \propto E}\langle\underline{\pi}\rangle \qquad \underline{s} := C_s\langle\overline{t}\rangle$$

The following lemmas sum up the properties of the decoding.

Lemma 7 (Closed Scopes Disappear). *Let $F \propto E$ be a compatible pair. Then $\underline{F \propto (\blacktriangle x : E_{\mathrm{w}} : \blacktriangledown x : E)} = \underline{F \propto E}$.*

Lemma 8 (LO Decoding Invariant). *Let $s = F \mid \overline{u} \mid \pi \mid E \mid \varphi$ be a reachable state. Then $\underline{F \propto E}$ and C_s are LO contexts.*

Lemma 9 (Decoding and Structural Equivalence $\equiv$).

1. Stacks and Substitutions Commute*: if x does not occur free in π then $\underline{\pi}\langle t[x{\leftarrow}u]\rangle \equiv \underline{\pi}\langle t\rangle[x{\leftarrow}u]$;*
2. Compatible Pairs Absorb Substitutions*: if x does not occur free in F then $\underline{F \propto E}\langle t[x{\leftarrow}u]\rangle \equiv \underline{F \propto ([x{\leftarrow}u] : E)}\langle t\rangle$.*

The next theorem is our first main result. By the abstract approach presented in Sect. 3 (Theorem 1), it implies that the Strong MAM is a correct and complete implementation of linear LO evaluation to normal form.

Theorem 3 (Distillation). *(Strong MAM, $\to_{\mathtt{LO}}, \equiv, \underline{\cdot}$) is an explicit and reflective distillery. In particular:*

1. Projection of Principal Transitions*:*
   (a) Multiplicative*: if $s \rightsquigarrow_{\mathtt{m}} s'$ then $\underline{s} \to_{\mathtt{m}} \equiv \underline{s'}$;*
   (b) Exponential*: if $s \rightsquigarrow_{\mathtt{e}} s'$ then $\underline{s} \to_{\mathtt{e}} \underline{s'}$, duplicating the same subterm.*
2. Distillation of Commutative Transitions*:*
   (a) Garbage Collection of Weak Environments*: if $s \rightsquigarrow_{\mathtt{c}_4} s'$ then $\underline{s} \equiv_{\mathtt{gc}} \underline{s'}$;*
   (b) Equality Cases*: if $s \rightsquigarrow_{\mathtt{c}_{1,2,3,5,6}} s'$ then $\underline{s} = \underline{s'}$.*

Proof. Recall, the decoding is defined as $\underline{(F, \overline{t}, \pi, E, \varphi)} := \underline{F \propto E}\langle\underline{\pi}\langle\overline{t}\rangle\rangle$. Determinism of the machine follows by the deterministic definition of $E(x)$, and that of the strategy follows from the totality of the LO order (Lemma 1). Transitions:

– **Case** $s = (F, \lambda x.\bar{t}, \overline{u} : \pi, E, \blacktriangledown) \rightsquigarrow_{\mathtt{m}} (F, \bar{t}, \pi, [x \leftarrow \overline{u}] : E, \blacktriangledown) = s'$**.** Note that $C_{s'} = \underline{F \propto E}\langle\underline{\pi}\rangle$ is LO by the LO decoding invariant (Lemma 8). Moreover by the closure invariant (Lemma 6.5) x does not occur in F nor π, justifying the use of Lemma 9 in:

$$\begin{array}{rll}
\underline{(F, \lambda x.\bar{t}, \overline{u} : \pi, E, \blacktriangledown)} = & & \underline{F \propto E}\langle\underline{\overline{u} : \pi}\langle\lambda x.\bar{t}\rangle\rangle \\
= & & \underline{F \propto E}\langle\underline{\pi}\langle(\lambda x.\bar{t})\overline{u}\rangle\rangle \\
\rightarrow_{\mathtt{m}} & & \underline{F \propto E}\langle\underline{\pi}\langle\bar{t}[x \leftarrow \overline{u}]\rangle\rangle \\
\equiv_{L.9.1} & & \underline{F \propto E}\langle\underline{\pi}\langle\bar{t}\rangle[x \leftarrow \overline{u}]\rangle \\
\equiv_{L.9.2} & & \underline{F \propto ([x \leftarrow \overline{u}] : E)}\langle\underline{\pi}\langle\bar{t}\rangle\rangle = \underline{(F, \bar{t}, \pi, [x \leftarrow \overline{u}] : E, \blacktriangledown)}
\end{array}$$

– **Case** $s = (F, x, \pi, E, \blacktriangledown) \rightsquigarrow_{\mathtt{e}} (F, \bar{t}^{\alpha}, \pi, E, \blacktriangledown) = s'$ **with** $E(x) = \bar{t}$**.** As before, C_s is LO by Lemma 8. Moreover, $E(x) = \bar{t}$ guarantees that E, and thus C_s, have a substitution binding x to $\bar{t}$. Finally, $C_s = C_{s'}$. Then

$$\underline{s} = C_s\langle x\rangle \rightarrow_{\mathtt{e}} C_s\langle\bar{t}^{\alpha}\rangle = \underline{s'}$$

– **Case** $s = (x : F, \bar{t}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle \mathtt{c}_4} (F, \lambda x.\bar{t}, \epsilon, \blacktriangle x : E, \blacktriangle) = s'$**.** By Lemma 6.1 $x : F \propto E$, and by Lemma 5.2 $E = E_{\mathrm{w}} : \blacktriangledown x : E'$. Then

$$\underline{(x : F) \propto E} = \underline{(x : F) \propto (E_{\mathrm{w}} : \blacktriangledown x : E')} = \underline{(x : F) \propto (\blacktriangledown x : E')}\langle\underline{E_{\mathrm{w}}}\rangle$$

Since we are in a backtracking phase ($\blacktriangle$), the backtracking free variables invariant (Lemma 6.3.1) and the open scopes matching property (Lemma 5.3) give $\mathtt{fv}(\bar{t}) \subseteq_{L.6.3.1} \Lambda(F) =_{L.5.3} \Lambda(E_{\mathrm{w}} : \blacktriangledown x : E') =_{L.4} \Lambda(\blacktriangledown x : E')$, *i.e.* $\underline{E_{\mathrm{w}}}$ does not bind any variable in $\mathtt{fv}(\bar{t})$. Then $\underline{E_{\mathrm{w}}}\langle\bar{t}\rangle \equiv^{*}_{\mathtt{gc}} \bar{t}$, and

$$\begin{array}{rll}
\underline{(x : F, \bar{t}, \epsilon, E, \blacktriangle)} = & & \underline{(x : F) \propto E}\langle\bar{t}\rangle \\
= & & \underline{(x : F) \propto (E_{\mathrm{w}} : \blacktriangledown x : E')}\langle\bar{t}\rangle \\
= & & \underline{(x : F) \propto (\blacktriangledown x : E')}\langle\underline{E_{\mathrm{w}}}\langle\bar{t}\rangle\rangle \\
\equiv^{*}_{\mathtt{gc}} & & \underline{(x : F) \propto (\blacktriangledown x : E')}\langle\bar{t}\rangle \\
= & & \underline{F \propto E'}\langle\lambda x.\bar{t}\rangle \\
=_{L.7} & & \underline{F \propto (\blacktriangle x : E_{\mathrm{w}} : \blacktriangledown x : E')}\langle\lambda x.\bar{t}\rangle \\
= & & \underline{F \propto (\blacktriangle x : E)}\langle\lambda x.\bar{t}\rangle \qquad = \underline{(F, \lambda x.\bar{t}, \epsilon, \blacktriangle x : E, \blacktriangle)}
\end{array}$$

– **Case** $(F, \bar{t}\overline{u}, \pi, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown \mathtt{c}_1} (F, \bar{t}, \overline{u} : \pi, E, \blacktriangledown)$**.**

$$\underline{(F, \bar{t}\overline{u}, \pi, E, \blacktriangledown)} = \underline{F \propto E}\langle\underline{\pi}\langle\bar{t}\overline{u}\rangle\rangle = \underline{F \propto E}\langle\underline{\overline{u} : \pi}\langle\bar{t}\rangle\rangle = \underline{(F, \bar{t}, \overline{u} : \pi, E, \blacktriangledown)}$$

– **Case** $(F, \lambda x.\bar{t}, \epsilon, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown \mathtt{c}_2} (x : F, \bar{t}, \epsilon, \blacktriangledown x : E, \blacktriangledown)$**.**

$$\begin{array}{rl}
\underline{(F, \lambda x.\bar{t}, \epsilon, E, \blacktriangledown)} = & \underline{F \propto E}\langle\lambda x.\bar{t}\rangle \\
= & \underline{(x : F) \propto (\blacktriangledown x : E)}\langle\bar{t}\rangle = \underline{(x : F, \bar{t}, \epsilon, \blacktriangledown x : E, \blacktriangledown)}
\end{array}$$

– **Case** $(F, x, \pi, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown \mathtt{c}_3} (F, x, \pi, E, \blacktriangle)$**.**

$$\underline{(F, x, \pi, E, \blacktriangledown)} = \underline{F \propto E}\langle\underline{\pi}\langle x\rangle\rangle = \underline{(F, x, \pi, E, \blacktriangle)}$$

– **Case** $((\bar{t},\pi) : F, \overline{u}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle \mathtt{c}_5} (F, \bar{t}\overline{u}, \pi, E, \blacktriangle)$**.**

$$\underline{((\bar{t},\pi) : F, \overline{u}, \epsilon, E, \blacktriangle)} = \underline{(\bar{t},\pi) : F \propto E\langle\overline{u}\rangle} = \underline{F \propto E\langle\underline{\pi}\langle\bar{t}\,\overline{u}\rangle\rangle} = \underline{(F, \bar{t}\overline{u}, \pi, E, \blacktriangle)}$$

– **Case** $(F, \bar{t}, \overline{u} : \pi, E, \blacktriangle) \rightsquigarrow_{\blacktriangle \mathtt{c}_6} ((\bar{t},\pi) : F, \overline{u}, \epsilon, E, \blacktriangledown)$**.**

$$\begin{aligned}\underline{(F, \bar{t}, \overline{u} : \pi, E, \blacktriangle)} &= \underline{F \propto E\langle\overline{u} : \pi\rangle\langle\bar{t}\rangle\rangle} \\ &= \underline{F \propto E\langle\underline{\pi}\langle\bar{t}\,\overline{u}\rangle\rangle} \\ &= \underline{((\bar{t},\pi) : F) \propto E\langle\overline{u}\rangle} = \underline{((\bar{t},\pi) : F, \overline{u}, \epsilon, E, \blacktriangledown)}\end{aligned}$$

For what concerns reflectiveness, *termination* of commutative transitions is subsumed by bilinearity (Theorem 4 below). For *progress*, note that

1. *the machine cannot get stuck during the evaluation phase*: for applications and abstractions it is evident and for variables one among $\rightsquigarrow_{\mathtt{e}}$ and $\rightsquigarrow_{\blacktriangledown \mathtt{c}_3}$ always applies, because of the closure invariant (Lemma 6.5).
2. *final states have the form* $(\epsilon, t, \epsilon, E, \blacktriangle)$, because
   (a) by the previous consideration they are in a backtracking phase,
   (b) if the stack is non-empty then $\rightsquigarrow_{\blacktriangle \mathtt{c}_6}$ applies,
   (c) otherwise if the frame is not empty then either $\rightsquigarrow_{\blacktriangle \mathtt{c}_4}$ or $\rightsquigarrow_{\blacktriangle \mathtt{c}_5}$ applies.
3. *final states decode to normal terms*: a final state $s = (\epsilon, t, \epsilon, E, \blacktriangle)$ decodes to $\underline{s} = \underline{E}\langle t\rangle$ which is normal and closed by the normal form (Lemma 6.2.1) and backtracking free variables (Lemma 6.3.1) invariants. □

7 Complexity Analysis

The complexity analysis requires a further invariant, bounding the size of the duplicated subterms. For us, $\overline{u}$ is a subterm of $\bar{t}$ if it does so up to variable names, both free and bound. More precisely: define t^- as t in which all variables (including those appearing in binders) are replaced by a fixed symbol $*$. Then, we will consider u to be a subterm of t whenever u^- is a subterm of t^- in the usual sense. The key property ensured by this definition is that the size $|\overline{u}|$ of $\overline{u}$ is bounded by $|\bar{t}|$.

Lemma 10 (Subterm Invariant). *Let ρ be an execution from an initial code $\bar{t}$. Every code duplicated along ρ using $\rightsquigarrow_{\mathtt{e}}$ is a subterm of $\bar{t}$.*

Via the distillation theorem (Theorem 3), the invariant provides a new proof of the subterm property of linear LO reduction (first proved in [6]).

Lemma 11 (Subterm Property for $\rightarrow_{\mathtt{LO}}$). *Let d be a $\rightarrow_{\mathtt{LO}}$-derivation from an initial term t. Every term duplicated along d using $\rightarrow_{\mathtt{e}}$ is a subterm of t.*

The next theorem is our second main result, from which the low-level implementation theorem (Theorem 2) follows. Let us stress that, despite the simplicity of the reasoning, the analysis is subtle as the length of backtracking phases (Point 2) can be bound only *globally*, by the whole previous evaluation work.

Theorem 4 (Bilinearity). *The Strong MAM is bilinear, i.e. given an execution $\rho : s \rightsquigarrow^* s'$ from an initial state of code t then:*

1. Commutative Evaluation Steps are Bilinear*:* $|\rho|_{\blacktriangledown \mathtt{c}} \leq (1 + |\rho|_{\mathtt{e}}) \cdot |t|$.
2. Commutative Evaluation Bounds Backtracking*:* $|\rho|_{\blacktriangle \mathtt{c}} \leq 2 \cdot |\rho|_{\blacktriangledown \mathtt{c}}$.
3. Commutative Steps are Bilinear*:* $|\rho|_c \leq 3 \cdot (1 + |\rho|_{\mathtt{e}}) \cdot |t|$.

Proof. 1. We prove a slightly stronger statement, namely $|\rho|_{\blacktriangledown \mathtt{c}} + |\rho|_{\mathtt{m}} \leq (1 + |\rho|_{\mathtt{e}}) \cdot |t|$, by means of the following notion of size for stacks/frames/states:

$$\begin{array}{rlrl} |\epsilon| &:= 0 & |x : F| &:= |F| \\ |\bar{t} : \pi| &:= |\bar{t}| + |\pi| & |(\bar{t}, \pi) : F| &:= |\pi| + |F| \\ |(F, \bar{t}, \pi, E, \blacktriangledown)| &:= |F| + |\pi| + |\bar{t}| & |(F, \bar{t}, \pi, E, \blacktriangle)| &:= |F| + |\pi| \end{array}$$

By direct inspection of the rules of the machine it can be checked that:

- *Exponentials Increase the Size*: if $s \rightsquigarrow_{\mathtt{e}} s'$ is an exponential transition, then $|s'| \leq |s| + |t|$ where $|t|$ is the size of the initial term; this is a consequence of the fact that exponential steps retrieve a piece of code from the environment, which is a subterm of the initial term by Lemma 10;
- *Non-Exponential Evaluation Transitions Decrease the Size*: if $s \rightsquigarrow_a s'$ with $a \in \{\mathtt{m}, \blacktriangledown \mathtt{c}_1, \blacktriangledown \mathtt{c}_2, \blacktriangledown \mathtt{c}_3\}$ then $|s'| < |s|$;
- *Backtracking Transitions do not Change the Size*: if $s \rightsquigarrow_a s'$ with $a \in \{\blacktriangle \mathtt{c}_4, \blacktriangle \mathtt{c}_5, \blacktriangle \mathtt{c}_6\}$ then $|s'| = |s|$.

Then a straightforward induction on $|\rho|$ shows that

$$|s'| \leq |s| + |\rho|_{\mathtt{e}} \cdot |t| - |\rho|_{\blacktriangledown \mathtt{c}} - |\rho|_{\mathtt{m}}$$

i.e. that $|\rho|_{\blacktriangledown \mathtt{c}} + |\rho|_{\mathtt{m}} \leq |s| + |\rho|_{\mathtt{e}} \cdot |t| - |s'|$.
Now note that $|\cdot|$ is always non-negative and that since s is initial we have $|s| = |t|$. We can then conclude with

$$\begin{array}{rll} |\rho|_{\blacktriangledown \mathtt{c}} + |\rho|_{\mathtt{m}} &\leq |s| + |\rho|_{\mathtt{e}} \cdot |t| - |s'| & \\ &\leq |s| + |\rho|_{\mathtt{e}} \cdot |t| & = |t| + |\rho|_{\mathtt{e}} \cdot |t| = (1 + |\rho|_{\mathtt{e}}) \cdot |t| \end{array}$$

2. We have to estimate $|\rho|_{\blacktriangle \mathtt{c}} = |\rho|_{\blacktriangle \mathtt{c}_4} + |\rho|_{\blacktriangle \mathtt{c}_5} + |\rho|_{\blacktriangle \mathtt{c}_6}$. Note that
   (a) $|\rho|_{\blacktriangle \mathtt{c}_4} \leq |\rho|_{\blacktriangledown \mathtt{c}_2}$, as $\rightsquigarrow_{\blacktriangle \mathtt{c}_4}$ pops variables from F, pushed only by $\rightsquigarrow_{\blacktriangledown \mathtt{c}_2}$;
   (b) $|\rho|_{\blacktriangle \mathtt{c}_5} \leq |\rho|_{\blacktriangle \mathtt{c}_6}$, as $\rightsquigarrow_{\blacktriangle \mathtt{c}_5}$ pops pairs $(\bar{t}, \pi)$ from F, pushed only by $\rightsquigarrow_{\blacktriangle \mathtt{c}_6}$;
   (c) $|\rho|_{\blacktriangle \mathtt{c}_6} \leq |\rho|_{\blacktriangledown \mathtt{c}_3}$, as $\rightsquigarrow_{\blacktriangle \mathtt{c}_6}$ ends backtracking phases, started only by $\rightsquigarrow_{\blacktriangledown \mathtt{c}_3}$.
   Then $|\rho|_{\blacktriangle \mathtt{c}} \leq |\rho|_{\blacktriangledown \mathtt{c}_2} + 2|\rho|_{\blacktriangledown \mathtt{c}_3} \leq 2|\rho|_{\blacktriangledown \mathtt{c}}$.
3. We have $|\rho|_c = |\rho|_{\blacktriangledown \mathtt{c}} + |\rho|_{\blacktriangle \mathtt{c}} \leq_{P.2} |\rho|_{\blacktriangledown \mathtt{c}} + 2|\rho|_{\blacktriangledown \mathtt{c}} =_{P.1} 3 \cdot (1 + |\rho|_{\mathtt{e}}) \cdot |t|$.

Last, every transition but $\rightsquigarrow_{\mathtt{e}}$ takes a constant time on a RAM.The renaming in a $\rightsquigarrow_{\mathtt{e}}$ step is instead linear in $|\bar{t}|$, by the subterm invariant (Lemma 10). □

Acknowledgments. This work was partially supported by projects Logoi ANR-2010-BLAN-0213-02, Coquas ANR-12-JS02-006-01, Elica ANR-14-CE25-0005, the Saint-Exupéry program funded by the French embassy and the Ministry of Education in Argentina, and the French–Argentinian laboratory in Computer Science INFINIS.

References

1. Abramsky, S., Ong, C.L.: Full abstraction in the lazy lambda calculus. Inf. Comput. 105(2), 159–267 (1993)
2. Accattoli, B.: An abstract factorization theorem for explicit substitutions. In: RTA. pp. 6–21 (2012)
3. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling abstract machines. In: ICFP. pp. 363–376 (2014)
4. Accattoli, B., Barenbaum, P., Mazza, D.: A strong distillery. CoRR abs/1509.00996 (2015), http://arxiv.org/abs/1509.00996
5. Accattoli, B., Bonelli, E., Kesner, D., Lombardi, C.: A nonstandard standardization theorem. In: POPL. pp. 659–670 (2014)
6. Accattoli, B., Dal Lago, U.: Beta Reduction is Invariant, Indeed. In: CSL-LICS. p. 8 (2014)
7. Accattoli, B., Sacerdoti Coen, C.: On the value of variables. In: WoLLIC 2014. pp. 36–50 (2014)
8. Accattoli, B., Sacerdoti Coen, C.: On the relative usefulness of fireballs. In: LICS. pp. 141–155 (2015)
9. Ariola, Z.M., Bohannon, A., Sabry, A.: Sequent calculi and abstract machines. ACM Trans. Program. Lang. Syst. 31(4) (2009)
10. Biernacka, M., Danvy, O.: A concrete framework for environment machines. ACM Trans. Comput. Log. 9(1) (2007)
11. Boutiller, P.: De nouveaus outils pour manipuler les inductif en Coq. Ph.D. thesis, Université Paris Diderot - Paris 7 (2014)
12. de Carvalho, D.: Execution time of lambda-terms via denotational semantics and intersection types. CoRR abs/0905.4251 (2009)
13. Crégut, P.: An abstract machine for lambda-terms normalization. In: LISP and Functional Programming. pp. 333–340 (1990)
14. Crégut, P.: Strongly reducing variants of the Krivine abstract machine. Higher-Order and Symbolic Computation 20(3), 209–230 (2007)
15. Curien, P.: An abstract framework for environment machines. Theor. Comput. Sci. 82(2), 389–402 (1991)
16. Danos, V., Regnier, L.: Head linear reduction (2004), unpublished
17. Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. Tech. Rep. RS-04-26, BRICS (2004)
18. Danvy, O., Zerny, I.: A synthetic operational account of call-by-need evaluation. In: PPDP. pp. 97–108 (2013)
19. Dénès, M.: Étude formelle d'algorithmes efficaces en algèbre linéaire. Ph.D. thesis, Université de Nice - Sophia Antipolis (2013)
20. Ehrhard, T., Regnier, L.: Böhm trees, Krivine's machine and the Taylor expansion of lambda-terms. In: CiE. pp. 186–197 (2006)
21. Fernández, M., Siafakas, N.: New developments in environment machines. Electr. Notes Theor. Comput. Sci. 237, 57–73 (2009)
22. García-Pérez, Á., Nogueira, P., Moreno-Navarro, J.J.: Deriving the full-reducing krivine machine from the small-step operational semantics of normal order. In: PPDP. pp. 85–96 (2013)
23. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: ICFP. pp. 235–246 (2002)
24. Hardin, T., Maranget, L.: Functional runtime systems within the lambda-sigma calculus. J. Funct. Program. 8(2), 131–176 (1998)

25. Lang, F.: Explaining the lazy Krivine machine using explicit substitution and addresses. Higher-Order and Symbolic Computation 20(3), 257–270 (2007)
26. Mascari, G., Pedicini, M.: Head linear reduction and pure proof net extraction. Theor. Comput. Sci. 135(1), 111–137 (1994)
27. Milner, R.: Local bigraphs and confluence: Two conjectures. Electr. Notes Theor. Comput. Sci. 175(3), 65–73 (2007)
28. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. 1(2), 125–159 (1975)
29. 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)
30. Smith, C.: Abstract machines for higher-order term sharing, Presented at IFL 2014