Distilling Abstract Machines

Beniamino Accattoli
Carnegie Mellon University & Università di Bologna
beniamino.accattoli@gmail.com

Pablo Barenbaum
University of Buenos Aires – CONICET
pbarenbaum@dc.uba.ar

Damiano Mazza
CNRS, UMR 7030, LIPN, Université Paris 13, Sorbonne Paris Cité
Damiano.Mazza@lipn.univ-paris13.fr

Abstract

It is well-known that many environment-based abstract machines can be seen as strategies in lambda calculi with explicit substitutions (ES). Recently, graphical syntaxes and linear logic led to the linear substitution calculus (LSC), a new approach to ES that is halfway between small-step calculi and traditional calculi with ES. This paper studies the relationship between the LSC and environment-based abstract machines. While traditional calculi with ES simulate abstract machines, the LSC rather distills them: some transitions are simulated while others vanish, as they map to a notion of structural congruence. The distillation process unveils that abstract machines in fact implement weak linear head reduction, a notion of evaluation having a central role in the theory of linear logic. We show that such a pattern applies uniformly in call-by-name, call-by-value, and call-by-need, catching many machines in the literature. We start by distilling the KAM, the CEK, and a sketch of the ZINC, and then provide simplified versions of the SECD, the lazy KAM, and Sestoft's machine. Along the way we also introduce some new machines with global environments. Moreover, we show that distillation preserves the time complexity of the executions, i.e. the LSC is a complexity-preserving abstraction of abstract machines.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming; F.1.1 [*Computation by Abstract Devices*]: Models of Computation; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages — Operational Semantics.; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic — Lambda Calculus and Related Systems.; I.1.3 [*Symbolic and Algebraic Manipulation*]: Languages and Systems — Evaluation Strategies

Keywords Lambda-calculus, abstract machines, explicit substitutions, linear logic, call-by-need, linear head reduction.

1. Introduction

In the theory of higher-order programming languages, abstract machines and explicit substitutions are two tools used to model the execution of programs on real machines while omitting many details of the actual implementation. Abstract machines can usually be seen as evaluation strategies in calculi of explicit substitutions (see at least [16, 19, 30, 36]), that can in turn be interpreted as cut-elimination strategies in sequent calculi [14].

Another tool providing a fine analysis of higher-order evaluation is linear logic, especially via the new perspectives on cut-elimination provided by *proof nets*, its graphical syntax. Explicit substitutions (ES) have been connected to linear logic by Kesner and co-authors in a sequence of works [26, 32, 33], culminating in the *linear substitution calculus* (LSC), a new formalism with ES behaviorally isomorphic to proof nets (introduced in [6], developed in [1, 3, 4, 7, 10], and bearing similarities with calculi by De Bruijn [25], Nederpelt [42], and Milner [41]). Since linear logic can model all evaluation schemes (call-by-name/value/need) [39], the LSC can express them modularly, by minor variations on rewriting rules and evaluation contexts. In this paper we revisit the relationship between environment-based abstract machines and ES. Traditionally, calculi with ES simulate machines. The LSC, instead, distills them.

A Bird's Eye View. In a simulation, every machine transition is simulated by some steps in the calculus with ES. In a distillation—a concept which we will formally define in the paper—only some of the machine transitions are simulated, while the others are mapped to the structural equivalence of the calculus, a specific trait of the LSC. Such an equivalence has a useful property: it commutes with evaluation, *i.e.* it can be postponed. Thus, the transitions mapped to the structural congruence fade away, without compromising the result of evaluation. Additionally, we show that machine executions and their distilled representation in the LSC have the same asymptotic length, *i.e.* the distillation process preserves the complexity of evaluation. The main point is that the LSC is arguably simpler than abstract machines, and also—as we will show—it can uniformly represent and decompose many different machines in the literature.

Traditional vs Contextual ES. Traditional calculi with ES (see [31] for a survey) implement β-reduction $(\lambda x.t)u \to_\beta t\{x \leftarrow u\}$ introducing an annotation (the explicit substitution $[x \leftarrow u]$),

$$(\lambda x.t)u \quad \to_{\mathtt{B}} \quad t[x \leftarrow u]$$

and percolating it through the term structure,

$$\begin{array}{rcl} (tw)[x \leftarrow u] & \to_{@} & t[x \leftarrow u]w[x \leftarrow u] \\ (\lambda x.t)[y \leftarrow u] & \to_{\lambda} & \lambda x.t[y \leftarrow u] \end{array} \tag{1}$$

until they reach variable occurrences on which they finally substitute or get garbage collected,

$$\begin{array}{rcl} x[x \leftarrow u] & \to_{\mathtt{var}} & u \\ y[x \leftarrow u] & \to_{\not\equiv} & y \end{array}$$

The LSC, instead, is based on a *contextual* view of evaluation and substitution, also known as *substitution at a distance*. The idea is that one can get rid of the rules percolating through the term structure—*i.e.* @ and λ—by introducing contexts C (i.e. terms with

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-2873-9/14/09…$15.00.
http://dx.doi.org/10.1145/2628136.2628154

a hole $\langle\cdot\rangle$) and generalizing the base cases, obtaining just two rules, *linear substitution* (ls) and *garbage collection* (gc):

$$\begin{array}{lll} C\langle x\rangle[x\leftarrow u] & \to_{\mathtt{ls}} & C\langle u\rangle[x\leftarrow u] \\ t[x\leftarrow u] & \to_{\mathtt{gc}} & t \qquad \text{if } x\notin \mathtt{fv}(t) \end{array}$$

Dually, the rule creating substitutions (B) is generalized to act up to a context of substitutions $[\ldots\leftarrow\ldots] := [x_1\leftarrow w_1]\ldots[x_k\leftarrow w_k]$ obtaining rule dB (B at a distance):

$$(\lambda x.t)[\ldots\leftarrow\ldots]u \quad \to_{\mathtt{dB}} \quad t[x\leftarrow u][\ldots\leftarrow\ldots]$$

Logical Perspective on the LSC. From a sequent calculus point of view, rules @ and λ, corresponding to *commutative* cut-elimination cases, are removed and integrated—via the use of contexts—directly in the definition of the *principal* cases B, var and $\neq$, obtaining the contextual rules dB, ls, and gc. This is analogous, at the level of terms, to the removal of commutative cases provided by proof nets (see [2] for a discussion about commutative cases and proof nets). From a linear logic point of view, $\to_{\mathtt{dB}}$ can be identified with the multiplicative cut-elimination case $\to_{\mathtt{m}}$, while $\to_{\mathtt{ls}}$ and $\to_{\mathtt{gc}}$ correspond to exponential cut-elimination. Actually, garbage collection has a special status, as it can always be postponed. We will then identify exponential cut-elimination $\to_{\mathtt{e}}$ with linear substitution $\to_{\mathtt{ls}}$ alone.

The LSC has a simple meta-theory, and is halfway between traditional calculi with ES—with whom it shares the micro-step dynamics—and λ-calculus—of which it retains most of the simplicity.

Distilling Abstract Machines. Abstract machines implement the traditional approach to ES, by

1. *Weak Evaluation*: forbidding reduction under abstraction (no rule $\to_\lambda$ in (1)),
2. *Evaluation Strategy*: looking for redexes according to some notion of weak evaluation context E,
3. *Context Representation*: using environments e (aka lists of substitutions) and stacks π (lists of terms) to keep track of the current evaluation context.

The LSC factorizes abstract machines. The idea is that one can represent the strategy of a machine by directly plugging the evaluation context in the contextual substitution/exponential rule:

$$E\langle x\rangle[x\leftarrow u] \quad \overset{E}{\to}_{\mathtt{e}} \quad E\langle u\rangle[x\leftarrow u]$$

factoring out the parts of the machine that just look for the next redex to reduce. By defining $\multimap$ as the closure of $\overset{E}{\to}_{\mathtt{e}}$ and $\to_{\mathtt{m}}$ by evaluation contexts E, one gets a clean representation of the machine strategy.

The mismatch between the two frameworks is in rule $\to_{@}$, that contextually—by nature—cannot be captured. In order to get out of this *cul-de-sac*, the very idea of *simulation* of an abstract machine must be refined. The crucial observation is that the equivalence $\equiv$ induced by $\to_{@}\cup\to_{\mathtt{gc}}$ has the same special status of $\to_{\mathtt{gc}}$, *i.e.* it can be postponed without affecting reduction lengths. More abstractly, $\equiv$ is a *strong bisimulation* with respect to $\multimap$, *i.e.* it verifies (note *one* step to *one* step, and *vice versa*)

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

and

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

These diagrams allow us to take $\equiv$ as a *structural equivalence* on the language. Indeed, the strong bisimulation property states that the transformation expressed by $\equiv$ is irrelevant with respect to $\multimap$, in particular $\equiv$-equivalent terms have $\multimap$-evaluations of the same length ending in $\equiv$-equivalent terms (and this holds even locally).

Abstract machines then are *distilled*: the logically relevant part of the substitution process is retained by $\multimap$ while both the search of the redex $\to_{@}$ and garbage collection $\to_{\neq}$ are isolated into the equivalence $\equiv$. Essentially, $\multimap$ captures principal cases of cut-elimination while $\equiv$ encapsulate the commutative ones (plus garbage collection, corresponding to principal cut-elimination involving weakenings).

Case Studies. We will analyze along these lines many abstract machines. Some are standard (KAM [34], CEK [28], a sketch of the ZINC [37]), some are new (MAM, MAD), and of others we provide simpler versions (SECD [35], Lazy KAM [19, 24], Sestoft's [44]). The previous explanation is a sketch of the distillation of the KAM, but the approach applies *mutatis mutandis* to all the other machines, encompassing most incarnations of call-by-name, call-by-value, and call-by need evaluation. The main contribution of the paper is indeed a modular *contextual* theory of abstract machines. We start by distilling some standard cases, and then show how the contextual view allows to understand and simplify non-trivial machines as the SECD, the lazy KAM, and Sestoft's abstract machine for call-by-need (deemed SAM). Our analysis enlightens their mechanisms as different and modular encodings of evaluation contexts for the LSC.

Call-by-Need. Along the way, we show that the contextual (or *at a distance*) approach of the LSC naturally leads to simple machines with just one global environment, as the newly introduced MAM (M for Milner). Such a feature is then showed to be a key ingredient of call-by-need machines, by using it to introduce a new and simple call-by-need machine, the MAD (the MAM by-neeD), and then showing how to obtain (simplifications of) the Lazy KAM and the SAM by simple tweaks. Morally, the global environment is a store. The contextual character of the LSC, however, models it naturally, without the need of extending the language with references.

Distillation Preserves Complexity. It is natural to wonder what is lost in the distillation process. What is the asymptotic impact of distilling machine executions into $\multimap$? Does it affect in any way the complexity of evaluation? We will show that *nothing is lost*, as machine executions are only linearly longer than $\multimap$. More precisely, they are *bilinear*, *i.e.* they are linear in 1) the length of $\multimap$, and in 2) the size $|t|$ of the starting term t. In other words, the search of redexes and garbage collection can be safely ignored in quantitative (time) analyses, *i.e.* the LSC and $\multimap$ provide a complexity-preserving abstraction of abstract machines. While in call-by-name and call-by-value such an analysis follows from an easy local property of machine executions, the call-by-need case is subtler, as such a local property does not hold and bilinearity can be established only via a global analysis.

Linear Logic and Weak Linear Head Reduction. Beyond the contextual view, our work also unveils a deep connection between abstract machines and linear logic. The strategies modularly encoding the various machines (generically noted $\multimap$ and parametric in a fixed notion of evaluation contexts) are in fact call-by-name/value/need versions of *weak linear head reduction* (WLHR), a fundamental notion in the theory of linear logic [3, 18, 21, 27, 40]. This insight is originally due to Danos and Regnier, who worked it out for the KAM [20]. Here, we develop it in a simpler and tighter way, modularly lifting it to many other abstract machines.

Call-by-Name. The call-by-name case (catching the KAM and the newly introduced MAM) is in fact special, as our distillation

theorem has three immediate corollaries, following from results about WLHR in the literature:

1. *Invariance*: it implies that the length of a KAM/MAM execution is an invariant time cost model (*i.e.* polynomially related to, say, Turing machines, in both directions), given that in [4] the same is shown for WLHR.
2. *Evaluation as Communication*: we implicitly establish a link between the KAM/MAM and the π-calculus, given that the evaluation of a term via WLHR is isomorphic to evaluation via Milner's encoding in the π-calculus [3].
3. *Plotkin's Approach*: our study complements the recent [10], where it is shown that WLHR is a standard strategy of the LSC. The two works together provide the lifting to explicit substitutions of Plotkin's approach of relating a machine (the SECD machine in that case, the KAM/MAM in ours) and a calculus (the call-by-value λ-calculus and the LSC, respectively) via a standardization theorem and a standard strategy [43].

Beyond Abstract Machines. This paper is just an episode—about abstract machines—in a recent feuilleton about complexity analysis of functional languages via linear logic and rewriting theory, starring the LSC. The story continues in [5] and [8]. In [5], the LSC is used to prove that the length of leftmost-outermost β-reduction is an invariant cost-model for λ-calculus (*i.e.* it is a measure polynomially related to evaluation in classic computational models like Turing machines or random access machines), solving a long-standing open problem in the theory of λ-calculus. Instead, [8] studies the asymptotic number of exponential steps (for $\multimap$) in terms of the number of multiplicative steps, in the call-by-name/value/need LSC (that is quadratic for call-by-name and linear for call-by-value/need). Via the results presented here, [8] establishes a polynomial relationship between the exponential and the multiplicative transitions of abstract machines, complementing our work.

Related Work. Beyond the already cited works, Danvy and coauthors have studied abstract machines in a number of works. In some of them, they show how to extract an abstract machine from a functional evaluator via a sequence of transformations (closure conversion, CPS, and defunctionalization) [11, 12, 22]. Such a study is orthogonal in spirit to what we do here. The only point of contact is the *rational deconstruction of the SECD* in [22], that is something that we also do, but in an orthogonal and less accurate way. Another sequence of works studies the relationship between abstract machines and calculi with ES [15, 16, 24], and it is clearly closer to our topic, except that: 1) [15, 16] follow the traditional (rather than the contextual) approach to ES; 2) none of these works deals with complexity analysis nor with linear logic. On the other hand, [16] provides a deeper analysis of Leroy's ZINC machine, as ours does not account for the avoidance of needless closure creations that is a distinct feature of the ZINC, and [24] focuses on the distinction between store-based and storeless call-by-need, a distinction that we address only implicitly (the calculus is storeless, but—as it will be discussed along the paper—it is meant to be implemented with a store). Last, what here we call *commutative transitions* essentially corresponds to what Danvy and Nielsen call *decompose* phase in [23].

The call-by-need calculus we use—that is a contextual reformulation of Maraist, Odersky, and Wadler's calculus [38]—is a novelty of this paper. It is simpler than both Ariola and Felleisen's [13] and Maraist, Odersky, and Wadler's calculi because it does not need any re-association axioms. A similar calculus is used by Danvy and Zerny in [24]. Morally, it is a version with let-bindings (avatars of ES) of Chang and Felleisen's calculus [17]. In [29], Garcia, Lumsdaine and Sabry present a further call-by-need machine, with whom we do not deal with.

Proofs. Some proofs have been omitted for lack of space. They can be found in the longer version [9].

2. Preliminaries on the Linear Substitution Calculus

Terms and Contexts. The language of the *weak linear substitution calculus* (WLSC) is generated by the following grammar:

$$t, u, w, r, q, p \quad ::= \quad x \mid v \mid tu \mid t[x{\leftarrow}u] \qquad\qquad v \quad ::= \lambda x.t$$

The constructor $t[x{\leftarrow}u]$ is called an *explicit substitution* (of u for x in t). The usual (implicit) substitution is instead denoted by $t\{x{\leftarrow}u\}$. Both $\lambda x.t$ and $t[x{\leftarrow}u]$ bind x in t, with the usual notion of α-equivalence. Values, noted v, do not include variables: this is a standard choice in the study of abstract machines, whose impact is analyzed in the companion paper [8].

Contexts are terms with one occurrence of the hole $\langle\cdot\rangle$, an additional constant. We will use many different contexts. The most general ones will be *weak contexts* W (*i.e.* not under abstractions), which are defined by:

$$W, W' \quad ::= \quad \langle\cdot\rangle \mid Wu \mid tW \mid W[x{\leftarrow}u] \mid t[x{\leftarrow}W]$$

The *plugging* $W\langle t\rangle$ (resp. $W\langle W'\rangle$) of a term t (resp. context W') in a context W is defined as $\langle t\rangle := t$ (resp. $\langle W'\rangle := W'$), $(Wt)\langle u\rangle := W\langle u\rangle t$ (resp. $(Wt)\langle W'\rangle := W\langle W'\rangle t$), and so on. The set of free variables of a term t (or context W) is denoted by $\mathtt{fv}(t)$ (resp. $\mathtt{fv}(W)$). Plugging in a context may capture free variables (replacing holes on the left of substitutions). These notions will be silently extended to all the contexts used in the paper.

Rewriting Rules. On the above terms, one may define several variants of the LSC by considering two elementary rewriting rules, *distance-β* (`dB`) and *linear substitution* (`ls`), each one coming in two variants, call-by-name and call-by-value (the latter variants being abbreviated by `dBv` and `lsv`), and pairing them in different ways and with respect to different evaluation contexts.

The rewriting rules rely in multiple ways on contexts. We start by defining *substitution contexts*, generated by

$$L \quad ::= \langle\cdot\rangle \mid L[x{\leftarrow}t].$$

A term of the form $L\langle v\rangle$ is an *answer*. Given a family of contexts C, the two variants of the elementary rewriting rules, also called *root rules*, are defined as follows:

$$\begin{array}{rcl} L\langle\lambda x.t\rangle u & \mapsto_{\mathtt{dB}} & L\langle t[x{\leftarrow}u]\rangle \\ L\langle\lambda x.t\rangle L'\langle v\rangle & \mapsto_{\mathtt{dBv}} & L\langle t[x{\leftarrow}L'\langle v\rangle]\rangle \\ C\langle x\rangle[x{\leftarrow}u] & \mapsto_{\mathtt{ls}} & C\langle u\rangle[x{\leftarrow}u] \\ C\langle x\rangle[x{\leftarrow}L\langle v\rangle] & \mapsto_{\mathtt{lsv}} & L\langle C\langle v\rangle[x{\leftarrow}v]\rangle \end{array}$$

In the linear substitution rules, we assume that $x \in \mathtt{fv}(C\langle x\rangle)$, *i.e.*, the context C does not capture the variable x, and we also silently work modulo α-equivalence to avoid variable capture in the rewriting rules. Moreover, we use the notations $\overset{C}{\mapsto}_{\mathtt{ls}}$ and $\overset{C}{\mapsto}_{\mathtt{lsv}}$ to specify the family of contexts used by the rules, with C being the meta-variable ranging over such contexts.

All of the above rules are *at a distance* (or *contextual*) because their definition involves contexts. Distance-β and linear substitution correspond, respectively, to the so-called *multiplicative* and *exponential* rules for cut-elimination in proof nets. The presence of contexts is how locality on proof nets is reflected on terms.

The rewriting rules decompose the usual small-step semantics for λ-calculi, by substituting one occurrence at the time, and only when such an occurrence is in evaluation position. We emphasize this fact saying that we adopt a *micro-step semantics*.

Calculus	Evaluation contexts	$\mapsto_{\mathtt{m}}$	$\mapsto_{\mathtt{e}}$	$\multimap_{\mathtt{m}}$	$\multimap_{\mathtt{e}}$
`Name`	$H ::= \langle\cdot\rangle \mid Ht \mid H[x{\leftarrow}t]$	$\mapsto_{\mathtt{dB}}$	$\overset{H}{\mapsto}_{\mathtt{ls}}$	$H\langle\mapsto_{\mathtt{dB}}\rangle$	$H\langle\overset{H}{\mapsto}_{\mathtt{ls}}\rangle$
`Value`$^{\mathrm{LR}}$	$V ::= \langle\cdot\rangle \mid Vt \mid L\langle v\rangle V \mid V[x{\leftarrow}t]$	$\mapsto_{\mathtt{dBv}}$	$\overset{V}{\mapsto}_{\mathtt{lsv}}$	$V\langle\mapsto_{\mathtt{dB}}\rangle$	$V\langle\overset{V}{\mapsto}_{\mathtt{ls}}\rangle$
`Value`$^{\mathrm{RL}}$	$S ::= \langle\cdot\rangle \mid SL\langle v\rangle \mid tS \mid S[x{\leftarrow}t]$	$\mapsto_{\mathtt{dBv}}$	$\overset{S}{\mapsto}_{\mathtt{lsv}}$	$S\langle\mapsto_{\mathtt{dB}}\rangle$	$S\langle\overset{S}{\mapsto}_{\mathtt{ls}}\rangle$
`Need`	$N ::= \langle\cdot\rangle \mid Nt \mid N[x{\leftarrow}t] \mid N'\langle x\rangle[x{\leftarrow}N]$	$\mapsto_{\mathtt{dB}}$	$\overset{N}{\mapsto}_{\mathtt{lsv}}$	$N\langle\mapsto_{\mathtt{dB}}\rangle$	$N\langle\overset{N}{\mapsto}_{\mathtt{ls}}\rangle$

Table 1. The four linear substitution calculi.

A linear substitution calculus is defined by a choice of root rules, *i.e.*, one of `dB`/`dBv` and one of `ls`/`lsv`, and a family of *evaluation contexts*. The chosen distance-β (resp. linear substitution) root rule is generically denoted by $\mapsto_{\mathtt{m}}$ (resp. $\mapsto_{\mathtt{e}}$). If E ranges over a fixed notion of evaluation context, the context-closures of the root rules are denoted by $\multimap_{\mathtt{m}} := E\langle\mapsto_{\mathtt{m}}\rangle$ and $\multimap_{\mathtt{e}} := E\langle\mapsto_{\mathtt{e}}\rangle$, where `m` (resp. `e`) stands for *multiplicative* (*exponential*). The rewriting relation defining the calculus is then $\multimap := \multimap_{\mathtt{m}} \cup \multimap_{\mathtt{e}}$.

2.1 Calculi

We consider four calculi, noted `Name`, `Value`$^{\mathrm{LR}}$, `Value`$^{\mathrm{RL}}$, and `Need`, and defined in Tab. 1. They correspond to four standard evaluation strategies for functional languages. We are actually slightly abusing the terminology, because—as we will show—they are *deterministic* calculi and thus should be considered as strategies. Our abuse is motivated by the fact that they are not strategies in the same calculus. The essential property of all these four calculi is that they are deterministic, because they implement a reduction strategy.

Proposition 2.1 (Determinism)**.** *The reduction relations of the four calculi of Tab. 1 are deterministic: in each calculus, if E_1, E_2 are evaluation contexts and if r_1, r_2 are redexes (i.e., terms matching the left hand side of the root rules defining the calculus), $E_1\langle r_1\rangle = E_2\langle r_2\rangle$ implies $E_1 = E_2$ and $r_1 = r_2$, so that there is at most one way to reduce a term, if any.*

Proof. See [9]. □

Call-by-Name (CBN). The evaluation contexts H for `Name` (defined in Tab. 1) are called *weak head contexts* and—when paired with micro-step evaluation—implement a strategy known as *weak linear head reduction*. The original presentation of this strategy does not use explicit substitutions [20, 40]. The presentation in use here has already appeared in [3, 10] (see also [1, 4]) as the weak head strategy of the *linear substitution calculus* (which is obtained by considering *all* contexts as evaluation contexts), and it avoids many technicalities of the original one. In particular, its relationship with the KAM is extremely natural, as we will show.

Let us give some examples of evaluation. Let $\delta := \lambda x.(xx)$ and consider the usual diverging term $\Omega := \delta\delta$. In `Name` it evaluates—diverging—as follows:

$$\begin{array}{lll}
\delta\delta = (\lambda x.(xx))\delta & \multimap_{\mathtt{m}} & (xx)[x{\leftarrow}\delta] \quad\quad\quad\quad\quad\quad\quad\quad \multimap_{\mathtt{e}} \\
& & (\delta x)[x{\leftarrow}\delta] \quad\quad\quad\quad\quad\quad\quad\quad \multimap_{\mathtt{m}} \\
& & (yy)[y{\leftarrow}x][x{\leftarrow}\delta] \quad\quad\quad\quad \multimap_{\mathtt{e}} \\
& & (xy)[y{\leftarrow}x][x{\leftarrow}\delta] \quad\quad\quad\quad \multimap_{\mathtt{e}} \\
& & (\delta y)[y{\leftarrow}x][x{\leftarrow}\delta] \quad\quad\quad\quad \multimap_{\mathtt{m}} \\
& & (zz)[z{\leftarrow}y][y{\leftarrow}x][x{\leftarrow}\delta] \quad\quad \multimap_{\mathtt{e}} \quad \ldots
\end{array}$$

Observe that according to our definitions both $\lambda x.\Omega$ and $x\Omega$ are $\multimap$-normal for `Name`, because evaluation does not go under abstractions, nor on the right of a variable (but terms like $x\Omega$ will be forbidden, as we will limit ourselves to closed terms). Now let us show the use of the context L in rule $\multimap_{\mathtt{m}}$. Let $I := \lambda y.y$ and $\tau := (\lambda z.\delta)I$, and consider the following variation over Ω, where rule $\multimap_{\mathtt{m}}$ is applied with $L := \langle\cdot\rangle[z{\leftarrow}I]$:

$$\begin{array}{lll}
\tau\tau = ((\lambda z.\delta)I)\tau & \multimap_{\mathtt{m}} & \delta[z{\leftarrow}I]\tau \quad\quad\quad\quad\quad\quad = \\
& & (\lambda x.(xx))[z{\leftarrow}I]\tau \quad \multimap_{\mathtt{m}} \\
& & (xx)[x{\leftarrow}\tau][z{\leftarrow}I] \quad \multimap_{\mathtt{e}} \quad \ldots
\end{array}$$

Call-by-Value (CBV). For CBV calculi (again see Tab. 1), *left-to-right* (`Value`$^{\mathrm{LR}}$) and *right-to-left* (`Value`$^{\mathrm{RL}}$) refer to the evaluation order of applications, *i.e.* they correspond to *operator first* and *argument first*, respectively (note the dual notions evaluation contexts V and S). The calculi `Value`$^{\mathrm{LR}}$ and `Value`$^{\mathrm{RL}}$ can be seen as strategies of a micro-step variant of the *value substitution calculus*, the (small-step) CBV calculus at a distance introduced in [7].

As an example, we consider again the evaluation of Ω. In `Value`$^{\mathrm{LR}}$ it goes as follows:

$$\begin{array}{lll}
\delta\delta = (\lambda x.(xx))\delta & \multimap_{\mathtt{m}} & (xx)[x{\leftarrow}\delta] \quad\quad\quad\quad\quad\quad\quad\quad \multimap_{\mathtt{e}} \\
& & (\delta x)[x{\leftarrow}\delta] \quad\quad\quad\quad\quad\quad\quad\quad \multimap_{\mathtt{e}} \\
& & (\delta\delta)[x{\leftarrow}\delta] \quad\quad\quad\quad\quad\quad\quad\quad \multimap_{\mathtt{m}} \\
& & (yy)[y{\leftarrow}\delta][x{\leftarrow}\delta] \quad\quad\quad\quad \multimap_{\mathtt{e}} \\
& & (\delta y)[y{\leftarrow}\delta][x{\leftarrow}\delta] \quad\quad\quad\quad \multimap_{\mathtt{e}} \ldots
\end{array}$$

While in `Value`$^{\mathrm{RL}}$ it takes the following form:

$$\begin{array}{lll}
\delta\delta = (\lambda x.(xx))\delta & \multimap_{\mathtt{m}} & (xx)[x{\leftarrow}\delta] \quad\quad\quad\quad\quad\quad\quad\quad \multimap_{\mathtt{e}} \\
& & (x\delta)[x{\leftarrow}\delta] \quad\quad\quad\quad\quad\quad\quad\quad \multimap_{\mathtt{e}} \\
& & (\delta\delta)[x{\leftarrow}\delta] \quad\quad\quad\quad\quad\quad\quad\quad \multimap_{\mathtt{m}} \\
& & (yy)[y{\leftarrow}\delta][x{\leftarrow}\delta] \quad\quad\quad\quad \multimap_{\mathtt{e}} \\
& & (y\delta)[y{\leftarrow}\delta][x{\leftarrow}\delta] \quad\quad\quad\quad \multimap_{\mathtt{e}} \ldots
\end{array}$$

Note that the CBV version of $\multimap_{\mathtt{m}}$ and $\multimap_{\mathtt{e}}$ employ substitution contexts L in a new way. An example of their use is given by the term $\tau\tau$ consider before for CBN. For instance, in `Value`$^{\mathrm{LR}}$:

$$\begin{array}{lll}
\tau\tau = ((\lambda z.\delta)I)\tau & \multimap_{\mathtt{m}} & \delta[z{\leftarrow}I]\tau \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \multimap_{\mathtt{m}} \\
& & \delta[z{\leftarrow}I](\delta[z{\leftarrow}I]) \quad\quad\quad\quad\quad\quad \multimap_{\mathtt{m}} \\
& & (xx)[x{\leftarrow}\delta[z{\leftarrow}I]][z{\leftarrow}I] \quad\quad \multimap_{\mathtt{e}} \\
& & (\delta x)[x{\leftarrow}\delta[z{\leftarrow}I]][z{\leftarrow}I] \quad\quad \ldots
\end{array}$$

Call-by-Need (CBNeed). The call-by-need calculus `Need` (Tab. 1) is a novelty of this paper, and can be seen either as a version at a distance of the calculi of [13, 38] or as a version with explicit substitution of the one in [17]. It fully exploits the fact that the two variants of the root rules may be combined: the β-rule is CBN, which reflects the fact that, operationally, the strategy is *by name*, but substitution is CBV, which forces arguments to be evaluated before being substituted, reflecting the *by need* content of the strategy. Please note the definition of CBNeed evaluation contexts N in Tab. 1. They extend the weak head contexts for CBN with a clause $(N'\langle x\rangle[x{\leftarrow}N])$ turning them into *hereditarily weak head contexts*. This new clause is how sharing is implemented by the reduction strategy. The general (non-deterministic) calculus is obtained by closing the root rules by *all* contexts, but its study is omitted. What we deal with here can be thought as its standard strategy (stopping on a sort of weak head normal form).

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

Figure 1. Axioms for structural equivalences. In $\equiv_{dup}$, $t_{[y]_x}$ denotes a term obtained from t by renaming some (possibly none) occurrences of x as y.

Let us show, once again, the evaluation of Ω an the impact of hereditarily head contexts. Consider:

$$
\begin{array}{lll}
\delta\delta = (\lambda x.(xx))\delta & \multimap_{\mathtt{m}} & (xx)[x{\leftarrow}\delta] \\
 & & (\delta x)[x{\leftarrow}\delta] \\
 & & (yy)[y{\leftarrow}x][x{\leftarrow}\delta] \\
 & & (yy)[y{\leftarrow}\delta][x{\leftarrow}\delta] \\
 & & (\delta y)[y{\leftarrow}\delta][x{\leftarrow}\delta] \\
 & & (zz)[z{\leftarrow}y][y{\leftarrow}\delta][x{\leftarrow}\delta]
\end{array}
\quad
\begin{array}{ll}
 & \\
\multimap_{\mathtt{e}} & \\
\multimap_{\mathtt{m}} & \\
\multimap_{\mathtt{e}} & \\
\multimap_{\mathtt{e}} & \\
\multimap_{\mathtt{m}} & \\
\multimap_{\mathtt{e}} & \ldots
\end{array}
$$

Note the difference with CBN in the second and fourth $\multimap_{\mathtt{e}}$ steps: the substitution rule replaces variable occurrences in explicit substitutions thanks to hereditarily weak evaluation contexts.

Structural equivalence. Another common feature of the four calculi is that they come with a notion of *structural equivalence*, denoted by $\equiv$. Consider Fig. 1. For CBN and CBV calculi, $\equiv$ is defined as the smallest equivalence relation containing the closure by weak contexts of $=_\alpha \cup \equiv_{gc} \cup \equiv_{dup} \cup \equiv_{@} \cup \equiv_{com} \cup \equiv_{[\cdot]}$ where $=_\alpha$ is α-equivalence. Call-by-need evaluates inside some substitutions (those hereditarily substituting on the head) and thus axioms as $\equiv_{dup}$ and $\equiv_{@}$ are too strong. Therefore, the structural equivalence for call-by-need, noted $\equiv_{\mathtt{Need}}$, is the one generated by $\equiv_{@l} \cup \equiv_{com} \cup \equiv_{[\cdot]}$.

Structural equivalence represents the fact that certain manipulations on explicit substitutions are computationally irrelevant, in the sense that they yield behaviorally equivalent terms. Technically, it is a *strong bisimulation* (the proof is in [9]):

Proposition 2.2 ($\equiv$ is a Strong Bisimulation)**.** *Let $\multimap_{\mathtt{m}}$, $\multimap_{\mathtt{e}}$ and $\equiv$ be the reduction relations and the structural equivalence relation of any of the calculi of Tab. 1, and let $\mathtt{x} \in \{\mathtt{m}, \mathtt{e}\}$. Then, $t \equiv u$ and $t \multimap_{\mathtt{x}} t'$ implies that there exists u' such that $u \multimap_{\mathtt{x}} u'$ and $t' \equiv u'$.*

The essential property of strong bisimulations is that they can be postponed. In fact, it is immediate to prove the following, which holds for all four calculi:

Lemma 2.3 ($\equiv$ Postponement)**.** *If $t \ (\multimap_{\mathtt{m}} \cup \multimap_{\mathtt{e}} \cup \equiv)^* \ u$ then $t \ (\multimap_{\mathtt{m}} \cup \multimap_{\mathtt{e}})^* \equiv u$ and the number of $\multimap_{\mathtt{m}}$ and $\multimap_{\mathtt{e}}$ steps in the two reduction sequences is exactly the same.*

In the simulation theorems for machines with a global environment (see Sect. 7.2 and Sect. 8) we will also use the following commutation property between substitutions and evaluation contexts via the structural equivalence of every evaluation scheme, proved by an easy induction on the actual definition of evaluation contexts.

Lemma 2.4 (ES Commute with Evaluation Contexts via $\equiv$)**.** *For every evaluation scheme let C denote an evaluation context s.t. $x \notin \mathtt{fv}(C)$ and $\equiv$ be its structural equivalence. Then $C\langle t\rangle[x{\leftarrow}u] \equiv C\langle t[x{\leftarrow}u]\rangle$.*

3. Preliminaries on Abstract Machines

Codes. All the abstract machines we will consider execute pure λ-terms. In our syntax, these are nothing but terms *without explicit substitutions*. Moreover, while for calculi we work implicitly modulo α, for machines we will *not* consider terms up to α, as the handling of α-equivalence characterizes different approaches to abstract machines. To stress these facts, we use the metavariables $\overline{t}, \overline{u}, \overline{w}, \overline{r}$ for pure λ-terms (not up to α) and $\overline{v}$ for pure values.

States. A machine state s will have various components, of which the first will always be *the code*, *i.e.* a pure λ-term $\overline{t}$. The others (*environment*, *stack*, *dump*,...) are all considered as lists, whose constructors are the empty list ϵ and the concatenation operator $::$. In fact, even if these components are formalized as lists, they may be intended to be implemented differently, as it will be the case for the machines with global environments (*i.e.* the MAM, the MAD, and its variants).

A state s of a machine is *initial* if its code $\overline{t}$ is closed (*i.e.*, $\mathtt{fv}(\overline{t}) = \varnothing$) and all other components are empty. An *execution* ρ is a sequence of transitions of the machine $s_0 \to^* s$ from an initial state s_0. In that case, we say that s is a *reachable state*, and if $\overline{t}$ is the code of s_0 then $\overline{t}$ is the *initial code* of s.

Invariants. For every machine our study will rely on a lemma about some *dynamic invariants*, *i.e.* some properties of the reachable states that are stable by executions. The lemma is always proved by a straightforward induction on the length of the execution and *the proof is omitted*.

Environments and Closures. There will be two types of machines, those with many *local environments* and those with just one *global environment*. Machines with local environments are based on the mutually recursive definition of *closure* (ranged over by c) and *environment* (e):

$$c \quad ::= \quad (\overline{t}, e) \qquad e \quad ::= \quad \epsilon \mid [x{\leftarrow}c] :: e$$

Global environments are defined by $E ::= \epsilon \mid [x{\leftarrow}\overline{t}] :: E$, and global environment machines will have just one global closure $(\overline{t}, E)$.

Well-Named and Closed Closures. The explicit treatment of α-equivalence, is based on particular representatives of α-classes defined via the notion of support. The *support* Δ of codes, environments, and closures is defined by:

- $\Delta(\overline{t})$ is the *multi*set of its bound names (*e.g.* $\Delta(\lambda x.\lambda y.\lambda x.(zx)) = [x, x, y]$).
- $\Delta(e)$ is the *multi*set of names captured by e (for example $\Delta([x{\leftarrow}c_1][y{\leftarrow}c_2][x{\leftarrow}c_3]) = [x, x, y]$), and similarly for $\Delta(E)$.
- $\Delta(\overline{t}, e) := \Delta(\overline{t}) + \Delta(e)$ and $\Delta(\overline{t}, E) := \Delta(\overline{t}) + \Delta(E)$.

A code/environment/closure $(\overline{t}, e)$ (resp. (t, E)) is *well-named* if its support $\Delta(\overline{t}, e)$ (resp. $\Delta(\overline{t}, E)$) is a set (*i.e.* a multiset with no repetitions). Moreover, a closure $(\overline{t}, e)$ (resp. (t, E)) is *closed* if $\mathtt{fv}(\overline{t}) \subseteq \Delta(e)$ (resp. $\mathtt{fv}(\overline{t}) \subseteq \Delta(E)$).

4. Distilleries

This section presents an abstract, high-level view of the relationship between abstract machines and linear substitution calculi, via the notion of *distillery* (see Tab. 2 for our pairs calculus/machine).

Definition 4.1. *A* distillery $\mathtt{D} = (\mathtt{M}, \mathtt{C}, \equiv, \underline{\cdot})$ *is given by:*

1. *An* abstract machine $\mathtt{M}$*, given by*
   (a) *a deterministic labeled transition system $\to$ on states s;*
   (b) *a distinguished class of states deemed* initial*, in bijection with closed λ-terms and from which one obtains the* reachable *states by applying $\to^*$;*
   (c) *a partition of the labels of the transition system $\to$ as:*

- commutative *transitions, noted* $\to_{\mathtt{c}}$*;*
- principal *transitions, in turn partitioned into*
  - multiplicative *transitions, denoted by* $\to_{\mathtt{m}}$*;*
  - exponential *transitions, denoted by* $\to_{\mathtt{e}}$*;*

2. *a* linear substitution calculus $\mathtt{C}$ *given by a pair* $(\multimap_{\mathtt{m}}, \multimap_{\mathtt{e}})$ *of rewriting relations on terms with ES;*
3. *a* structural equivalence $\equiv$ *on terms s.t. it is a strong bisimulation with respect to* $\multimap_{\mathtt{m}}$ *and* $\multimap_{\mathtt{e}}$*;*
4. *a* distillation $\underline{\cdot}$*, i.e. a decoding function from states to terms, s.t. on reachable states:*
   - Commutative*:* $s \to_{\mathtt{c}} s'$ *implies* $\underline{s} \equiv \underline{s'}$*.*
   - Multiplicative*:* $s \to_{\mathtt{m}} s'$ *implies* $\underline{s} \multimap_{\mathtt{m}} \equiv \underline{s'}$*;*
   - Exponential*:* $s \to_{\mathtt{e}} s'$ *implies* $\underline{s} \multimap_{\mathtt{e}} \equiv \underline{s'}$*;*

Given a distillery, the simulation theorem holds abstractly. Let $|\rho|$ (resp. $|d|$), $|\rho|_m$ (resp. $|d|_m$), $|\rho|_e$ (resp. $|d|_e$), and $|\rho|_p$ denote the number of unspecified, multiplicative, exponential, and principal steps in an execution (resp. derivation).

Theorem 4.2 (Simulation)**.** *Let* $\mathtt{D}$ *be a distillery. Then for every execution* $\rho : s \to^* s'$ *there is a derivation* $d : \underline{s} \multimap^* \equiv \underline{s'}$ *s.t.* $|\rho|_m = |d|_m$*,* $|\rho|_e = |d|_e$*, and* $|\rho|_p = |d|$*.*

Proof. By induction on $|\rho|$ and by the properties of the decoding, it follows that there is a derivation $e : \underline{s}(\multimap\equiv)^* \underline{s'}$ s.t. the number $|\rho|_p = |e|$. The witness d for the statement is obtained by applying the postponement of strong bisimulations (Lemma 2.3) to e. □

Reflection. Given a distillery, one would also expect that reduction in the calculus is reflected in the machine. This result in fact requires two additional abstract properties.

Definition 4.3 (Reflective Distillery)**.** *A distillery is* reflective *when:*

Termination: $\to_{\mathtt{c}}$ terminates *(on reachable states); hence, by determinism, every state* s *has a unique* commutative normal form $\mathtt{nf}_{\mathtt{c}}(s)$*;*

Progress: *if* s *is reachable,* $\mathtt{nf}_{\mathtt{c}}(s) = s$ *and* $\underline{s} \multimap_{\mathtt{x}} t$ *with* $\mathtt{x} \in \{\mathtt{m}, \mathtt{e}\}$*, then there exists* s' *such that* $s \to_{\mathtt{x}} s'$*, i.e.,* s *is not final.*

Then, we may prove the following reflection of steps in full generality:

Proposition 4.4 (Reflection)**.** *Let* $\mathtt{D}$ *be a reflective distillery,* s *be a reachable state, and* $\mathtt{x} \in \{\mathtt{m}, \mathtt{e}\}$*. Then,* $\underline{s} \multimap_{\mathtt{x}} u$ *implies that there exists a state* s' *s.t.* $\mathtt{nf}_{\mathtt{c}}(s) \to_{\mathtt{x}} s'$ *and* $\underline{s'} \equiv u$*.*

In other words, every rewriting step on the calculus can be also performed on the machine, up to commutative transitions.

Proof. The proof is by induction on the number n of transitions leading from s to $\mathtt{nf}_{\mathtt{c}}(s)$.

- *Base case* $n = 0$: by the progress property, we have $s \to_{\mathtt{x}'} s'$ for some state s' and $\mathtt{x}' \in \{\mathtt{m}, \mathtt{e}\}$. By Theorem 4.2, we have $\underline{s} \multimap_{\mathtt{x}'} u' \equiv \underline{s'}$ and we may conclude because $\mathtt{x}' = \mathtt{x}$ and $u' = u$ by determinism of the calculus (Proposition 2.1).
- *Inductive case* $n > 0$: by hypothesis, we have $s \to_{\mathtt{c}} s_1$. By Theorem 4.2, $\underline{s} \equiv \underline{s_1}$. The hypothesis and the strong bisimulation property (Proposition 2.2) then give us $\underline{s_1} \multimap_{\mathtt{x}} u_1 \equiv u$. But the induction hypothesis holds for s_1, giving us a state s' such that $\mathtt{nf}_{\mathtt{c}}(s_1) \to_{\mathtt{x}} s'$ and $\underline{s'} \equiv u_1 \equiv u$. We may now conclude because $\mathtt{nf}_{\mathtt{c}}(s) = \mathtt{nf}_{\mathtt{c}}(s_1)$. □

The reflection can then be extended to a reverse simulation.

Corollary 4.5 (Reverse Simulation)**.** *Let* $\mathtt{D}$ *be a reflective distillery and* s *an initial state. Given a derivation* $d : \underline{s} \multimap^* t$ *there is an execution* $\rho : s \to^* s'$ *s.t.* $t \equiv \underline{s'}$ *and* $|\rho|_m = |d|_m$*,* $|\rho|_e = |d|_e$*, and* $|\rho|_p = |d|$*.*

Calculus	*Abstract Machine*
$\mathtt{Name}$	KAM, MAM
$\mathtt{Value}^{\mathtt{LR}}$	CEK, Split CEK
$\mathtt{Value}^{\mathtt{RL}}$	LAM
$\mathtt{Need}$	(Merged/Pointing) MAD

Table 2. Correspondence between calculi of Tab. 1 and abstract machines.

Proof. By induction on the length of d, using Proposition 4.4. □

In the following sections we will introduce abstract machines and distillations for which we will prove that they form reflective distilleries with respect to the calculi of Sect. 2. For each machine we will prove: 1) that the decoding is in fact a distillation, and 2) the progress property. *We will instead assume the termination property*, whose proof is delayed to the quantitative study of the second part of the paper, where we will actually prove stronger results, giving explicit bounds.

5. Call-by-Name: the KAM

The Krivine Abstract Machine (KAM) is the simplest machine studied in the paper. A KAM *state* (s) is made out of a closure and of a *stack* (π):

$$\pi \quad ::= \quad \epsilon \mid c :: \pi \qquad\qquad s \quad ::= \quad (c, \pi)$$

For readability, we will use the notation $\overline{t} \mid e \mid \pi$ for a state (c, π) where $c = (\overline{t}, e)$. The transitions of the KAM then are:

$$\begin{array}{c|c|c|c|c|c|c} \overline{t}\overline{u} & e & \pi & \to_{\mathtt{c}} & \overline{t} & e & (\overline{u}, e) :: \pi \\ \lambda x.\overline{t} & e & c :: \pi & \to_{\mathtt{m}} & \overline{t} & [x \leftarrow c] :: e & \pi \\ x & e & \pi & \to_{\mathtt{e}} & \overline{t} & e' & \pi \end{array}$$

where $\to_{\mathtt{e}}$ takes place only if $e = e'' :: [x \leftarrow (\overline{t}, e')] :: e'''$.

A key point of our study is that environments and stacks rather immediately become contexts of the LSC, through the following decoding:

$$\begin{array}{rcl@{\qquad}rcl} \underline{\epsilon} & := & \langle\cdot\rangle & \underline{[x \leftarrow c] :: e} & := & \underline{e}\langle\langle\cdot\rangle[x \leftarrow \underline{c}]\rangle \\ \underline{(\overline{t}, e)} & := & \underline{e}\langle\overline{t}\rangle & \underline{c :: \pi} & := & \underline{\pi}\langle\langle\cdot\rangle\underline{c}\rangle \\ \underline{\overline{t} \mid e \mid \pi} & := & \underline{\pi}\langle\underline{e}\langle\overline{t}\rangle\rangle & & & \end{array}$$

The decoding satisfies the following static properties, shown by easy inductions on the definition.

Lemma 5.1 (Contextual Decoding)**.** *Let* e *be an environment and* π *be a stack of the KAM. Then* $\underline{e}$ *is a substitution context, and both* $\underline{\pi}$ *and* $\underline{\pi}\langle\underline{e}\rangle$ *are evaluation contexts.*

Next, we need the dynamic invariants of the machine.

Lemma 5.2 (KAM Invariants)**.** *Let* $s = \overline{u} \mid e \mid \pi$ *be a KAM reachable state whose initial code* $\overline{t}$ *is well-named. Then:*

1. Closure*: every closure in* s *is closed;*
2. Subterm*: any code in* s *is a literal subterm of* $\overline{t}$*.*
3. Name*: any closure* c *in* s *is well-named and its names are names of* $\overline{t}$ *(i.e.* $\Delta(c) \subseteq \mathtt{fv}(\overline{t})$*).*
4. Environment Size*: the length of any environment in* s *is bound by* $|\overline{t}|$*.*

Abstract Considerations on Concrete Implementations. The name invariant is the abstract property that allows to avoid both α-equivalence and name generation in KAM executions. Note that, by definition of well-named closure, there cannot be repetitions in the support of an environment. Then the length of any environment in any reachable state is bound by the number of distinct names in the initial code $\overline{t}$, *i.e.* with $|\overline{t}|$. This fact is important, as the static bound on the size of environments guarantees that $\to_{\mathtt{e}}$ and $\to_{\mathtt{c}}$—the transitions looking-up and copying environments—can be implemented (independently of the chosen concrete representation of terms) in at worst linear time in $|t|$, so that an execution ρ can be implemented in $O(|\rho| \cdot |t|)$. The same will hold for every machine with local environments. In fact, we may turn this into a definition: an abstract machine is *reasonable* if its implementation enjoys the above bilinear bound. In this way, the length of an execution of a reasonable machine provides an accurate estimate of its implementation cost.

The previous considerations are based on the name and environment size invariants. The closure invariant is used in the progress part of the next theorem, and the subterm invariant is used in the quantitative analysis in Sect. 11 (Theorem 11.3), subsuming the termination condition of reflective distilleries.

Theorem 5.3 (KAM Distillation). $(\mathit{KAM}, \mathtt{Name}, \equiv, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative*: if $s \to_{\mathtt{c}} s'$ then $\underline{s} \equiv \underline{s'}$.*
2. Multiplicative*: if $s \to_{\mathtt{m}} s'$ then $\underline{s} \multimap_{\mathtt{m}} \underline{s'}$;*
3. Exponential*: if $s \to_{\mathtt{e}} s'$ then $\underline{s} \multimap_{\mathtt{e}} \equiv \underline{s'}$;*

Proof. Properties of the decoding:

1. *Commutative*. We have $\overline{t}\overline{u} \mid e \mid \pi \to_{\mathtt{c}} \overline{t} \mid e \mid (\overline{u}, e) :: \pi$, and:

$$\begin{array}{rcl} \underline{\overline{t}\overline{u} \mid e \mid \pi} & = & \underline{\pi}\langle \underline{e}\langle \overline{t}\overline{u}\rangle\rangle \\ & \equiv^{*}_{@} & \underline{\pi}\langle \underline{e}\langle \overline{t}\rangle \underline{e}\langle \overline{u}\rangle\rangle \quad = \quad \underline{\overline{t} \mid e \mid (\overline{u}, e) :: \pi} \end{array}$$

2. *Multiplicative*. $\lambda x.\overline{t} \mid e \mid c :: \pi \to_{\mathtt{m}} \overline{t} \mid [x \leftarrow c] :: e \mid \pi$, and

$$\begin{array}{rcl} \underline{\lambda x.\overline{t} \mid e \mid c :: \pi} & = & \underline{\pi}\langle \underline{e}\langle \lambda x.\overline{t}\rangle \underline{c}\rangle \\ & \multimap_{\mathtt{m}} & \underline{\pi}\langle \underline{e}\langle \overline{t}[x \leftarrow \underline{c}]\rangle\rangle \\ & = & \underline{\overline{t} \mid [x \leftarrow c] :: e \mid \pi} \end{array}$$

   The rewriting step can be applied because by contextual decoding (Lemma 5.1) it takes place in an evaluation context.
3. *Exponential*. $x \mid e' :: [x \leftarrow (\overline{t}, e)] :: e'' \mid \pi \to_{\mathtt{e}} \overline{t} \mid e \mid \pi$, and

$$\begin{array}{rcl} \underline{x \mid e' :: [x \leftarrow (\overline{t}, e)] :: e'' \mid \pi} & = & \underline{\pi}\langle \underline{e''}\langle \underline{e'}\langle x\rangle [x \leftarrow \underline{e}\langle \overline{t}\rangle]\rangle\rangle \\ & \multimap_{\mathtt{e}} & \underline{\pi}\langle \underline{e''}\langle \underline{e'}\langle \underline{e}\langle \overline{t}\rangle\rangle [x \leftarrow \underline{e}\langle \overline{t}\rangle]\rangle\rangle \\ & \equiv^{*}_{gc} & \underline{\pi}\langle \underline{e}\langle \overline{t}\rangle\rangle \\ & = & \underline{\overline{t} \mid e \mid \pi} \end{array}$$

   Note that $e''\langle e'\langle e\langle \overline{t}\rangle\rangle [x \leftarrow e\langle \overline{t}\rangle]\rangle \equiv^{*}_{gc} e\langle \overline{t}\rangle$ holds because $e\langle \overline{t}\rangle$ is closed by point 1 of Lemma 5.2, and so all the substitutions around it can be garbage collected.

Termination. Given by (forthcoming) Theorem 11.3 (future proofs of distillery theorems will omit termination).
Progress. Let $s = \overline{t} \mid e \mid \pi$ be a commutative normal form s.t. $\underline{s} \multimap u$. If $\overline{t}$ is

- *an application $\overline{u}\overline{w}$*. Then a $\to_{\mathtt{c}}$ transition applies and s is not a commutative normal form, absurd;
- *an abstraction $\lambda x.\overline{u}$*: if $\pi = \epsilon$ then $\underline{s} = \underline{e}\langle \lambda x.\overline{u}\rangle$, which is $\multimap$-normal, absurd. Hence, a $\to_{\mathtt{m}}$ transition applies;
- *a variable x*: by point 1 of Lemma 5.2.1, we must have $e = e' :: [x \leftarrow c] :: e''$, so a $\to_{\mathtt{e}}$ transition applies. □

6. Call-by-Value: the CEK and the LAM

Here we deal with two adaptations to CBV of the KAM, namely Felleisen and Friedman's CEK machine [28] (without control operators), and a variant, deemed *Leroy Abstract Machine* (LAM). They differ in how they behave with respect to applications: the CEK implements left-to-right CBV, *i.e.* it first evaluates the function part, the LAM gives instead precedence to arguments, realizing right-to-left CBV. The LAM owes its name to Leroy's ZINC machine [37], that implements right-to-left CBV evaluation. We introduce a new name because the ZINC is a quite more sophisticated machine than the LAM: it has a separate sets of instructions to which terms are compiled, it handles arithmetic expressions, and it avoids needless closure creations in a way that it is not captured by the LAM. We deal with the LAM only to stress the modularity of our contextual approach.

CBV States and Stacks. The states of the CEK and the LAM have the same shape of those of the KAM, *i.e.* they are given by a closure plus a stack. The difference is that they use *CBV stacks*, whose elements are labelled either as *functions* or *arguments*, so that the machine may know whether it is launching the evaluation of an argument or it is at the end of such an evaluation. They are re-defined and decoded as follows (c is a closure):

$$\pi \quad ::= \quad \epsilon \mid \mathbf{f}(c) :: \pi \mid \mathbf{a}(c) :: \pi \qquad\qquad \begin{array}{rcl} \underline{\epsilon} & := & \langle\cdot\rangle \\ \underline{\mathbf{f}(c) :: \pi} & := & \underline{\pi}\langle \underline{c}\langle\cdot\rangle\rangle \\ \underline{\mathbf{a}(c) :: \pi} & := & \underline{\pi}\langle \langle\cdot\rangle \underline{c}\rangle \end{array}$$

The states of both machines are decoded exactly as for the KAM, *i.e.* $\underline{\overline{t} \mid e \mid \pi} := \underline{\pi}\langle \underline{e}\langle \overline{t}\rangle\rangle$.

6.1 Left-to Right Call-by-Value: the CEK machine.

The transitions of the CEK are:

$$\begin{array}{c|c|c|c|c|c|c|c} \overline{t}\overline{u} & e & \pi & \to_{\mathtt{c}_1} & \overline{t} & e & \mathbf{a}(\overline{u}, e) :: \pi \\ \overline{v} & e & \mathbf{a}(\overline{u}, e') :: \pi & \to_{\mathtt{c}_2} & \overline{u} & e' & \mathbf{f}(\overline{v}, e) :: \pi \\ \overline{v} & e & \mathbf{f}(\lambda x.\overline{t}, e') :: \pi & \to_{\mathtt{m}} & \overline{t} & [x \leftarrow (\overline{v}, e)] :: e' & \pi \\ x & e & \pi & \to_{\mathtt{e}} & \overline{t} & e' & \pi \end{array}$$

where $\to_{\mathtt{e}}$ takes place only if $e = e'' :: [x \leftarrow (\overline{t}, e')] :: e'''$.

While one can still statically prove that environments decode to substitution contexts, to prove that $\underline{\pi}$ and $\underline{\pi}\langle \underline{e}\rangle$ are evaluation contexts we need the dynamic invariants of the machine.

Lemma 6.1 (CEK Invariants). *Let $s = \overline{u} \mid e \mid \pi$ be a CEK reachable state whose initial code $\overline{t}$ is well-named. Then:*

1. Closure*: every closure in s is closed;*
2. Subterm*: any code in s is a literal subterm of $\overline{t}$;*
3. Value*: any code in e is a value and, for every element of π of the form $\mathbf{f}(\overline{u}, e')$, $\overline{u}$ is a value;*
4. Contextual Decoding*: $\underline{\pi}$ and $\underline{\pi}\langle \underline{e}\rangle$ are left-to-right CBV evaluation contexts.*
5. Name*: any closure c in s is well-named and its names are names of $\overline{t}$ (i.e. $\Delta(c) \subseteq \mathtt{fv}(\overline{t})$).*
6. Environment Size*: the length of any environment in s is bound by $|\overline{t}|$.*

We have everything we need:

Theorem 6.2 (CEK Distillation). $(\mathit{CEK}, \mathtt{Value}^{\mathtt{LR}}, \equiv, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative 1*: if $s \to_{\mathtt{c}_1} s'$ then $\underline{s} \equiv \underline{s'}$;*
2. Commutative 2*: if $s \to_{\mathtt{c}_2} s'$ then $\underline{s} = \underline{s'}$.*
3. Multiplicative*: if $s \to_{\mathtt{m}} s'$ then $\underline{s} \multimap_{\mathtt{m}} \underline{s'}$;*
4. Exponential*: if $s \to_{\mathtt{e}} s'$ then $\underline{s} \multimap_{\mathtt{e}} \equiv \underline{s'}$;*

$$\begin{array}{c|c|c|c|c|c|c|c|c} \overline{t}\overline{u} & e & \pi & D & \to_{\mathtt{c}_1} & \overline{t} & e & (\overline{u},e)::\pi & D \\ \overline{v} & e & (\overline{t},e')::\pi & D & \to_{\mathtt{c}_2} & \overline{t} & e' & \epsilon & ((\overline{v},e),\pi)::D \\ \overline{v} & e & \epsilon & ((\lambda x.\overline{t},e'),\pi)::D & \to_{\mathtt{m}} & \overline{t} & [x{\leftarrow}(\overline{v},e)]::e' & \pi & D \\ x & e::[x{\leftarrow}(\overline{v},e')]::e'' & \pi & D & \to_{\mathtt{e}} & \overline{v} & e' & \pi & D \end{array}$$

Figure 2. The Split CEK, aka the revisited SECD.

Proof. Properties of the decoding: in the following cases, evaluation will always takes place under a context that by Lemma 6.1.4 will be a left-to-right CBV evaluation context, and similarly structural equivalence will alway be used in a weak context, as it should be.

1. *Commutative 1*. We have $\overline{t}\overline{u} \mid e \mid \pi \to_{\mathtt{c}_1} \overline{t} \mid e \mid \mathbf{a}(\overline{u},e)::\pi$:

$$\begin{array}{rcll} \underline{\overline{t}\overline{u} \mid e \mid \pi} & = & \underline{\pi}\langle\underline{e}\langle\overline{t}\overline{u}\rangle\rangle & \equiv^*_{@} \\ & & \underline{\pi}\langle\underline{e}\langle\overline{t}\rangle\underline{e}\langle\overline{u}\rangle\rangle & = \quad \underline{\overline{t} \mid e \mid \mathbf{a}(\overline{u},e)::\pi} \end{array}$$

2. *Commutative 2*. We have $\overline{v} \mid e \mid \mathbf{a}(\overline{u},e') :: \pi \to_{\mathtt{c}_2} \overline{u} \mid e' \mid \mathbf{f}(\overline{v},e)::\pi$, and:

$$\begin{array}{rcll} \underline{\overline{v} \mid e \mid \mathbf{a}(\overline{u},e')::\pi} & = & \underline{\pi}\langle\underline{e}\langle\overline{v}\rangle\underline{e'}\langle\overline{u}\rangle\rangle & = \\ & & \underline{\overline{u} \mid e' \mid \mathbf{f}(\overline{v},e)::\pi} & \end{array}$$

3. *Multiplicative*. We have $\overline{v} \mid e \mid \mathbf{f}(\lambda x.\overline{t},e') :: \pi \to_{\mathtt{m}} \overline{u} \mid [x{\leftarrow}(\overline{v},e)]::e' \mid \pi$, and:

$$\begin{array}{rcll} \underline{\overline{v} \mid e \mid \mathbf{f}(\lambda x.\overline{t},e')::\pi} & = & \underline{\pi}\langle\underline{e'}\langle\lambda x.\overline{t}\rangle\underline{e}\langle\overline{v}\rangle\rangle & \multimap_{\mathtt{m}} \\ & & \underline{\pi}\langle\underline{e'}\langle\overline{t}[x{\leftarrow}\underline{e}\langle\overline{v}\rangle]\rangle\rangle & = \\ & & \underline{\overline{t} \mid [x{\leftarrow}(\overline{v},e)]::e' \mid \pi} & \end{array}$$

4. *Exponential*. Let $e = e'' :: [x{\leftarrow}(\overline{t},e')] :: e'''$. We have $x \mid e \mid \pi \to_{\mathtt{e}} \overline{t} \mid e' \mid \pi$, and:

$$\begin{array}{rcll} \underline{x \mid e \mid \pi} & = & \pi\langle e\langle x\rangle\rangle & = \\ & & \underline{\pi}\langle\underline{e'''}\langle e''\langle x\rangle[x{\leftarrow}\underline{e'}\langle\overline{t}\rangle]\rangle\rangle & \multimap_{\mathtt{e}} \\ & & \underline{\pi}\langle\underline{e'''}\langle\underline{e'}\langle e''\langle\overline{t}\rangle[x{\leftarrow}\overline{t}]\rangle\rangle\rangle & \equiv^*_{gc} \\ & & \underline{\pi}\langle\underline{e'}\langle\overline{t}\rangle\rangle & = \quad \underline{\overline{t} \mid e' \mid \pi} \end{array}$$

   We can apply $\multimap_{\mathtt{e}}$ since by Lemma 6.1.3, $\overline{t}$ is a value. We also use that by Lemma 6.1.1, $\underline{e'}\langle\overline{t}\rangle$ is a closed term to ensure that $\underline{e''}$ and $\underline{e'''}$ can be garbage collected.

Progress. Let $s = \overline{t} \mid e \mid \pi$ be a commutative normal form s.t. $\underline{s} \multimap u$. If $\overline{t}$ is

- *an application* $\overline{u}\overline{w}$. Then a $\to_{\mathtt{c}_1}$ transition applies and s is not a commutative normal form, absurd;
- *an abstraction* $\overline{v}$: by hypothesis, π cannot be of the form $\mathbf{a}(c) :: \pi'$. Suppose it is equal to ϵ. We would then have $\underline{s} = \underline{e}\langle\overline{v}\rangle$, which is a CBV normal form, because $\underline{e}$ is a substitution context. This would contradict our hypothesis, so π must be of the form $\mathbf{f}(\overline{u},e') :: \pi'$. By point 3 of Lemma 6.1, $\overline{u}$ is an abstraction, hence a $\to_{\mathtt{m}}$ transition applies;
- *a variable* x: by point 1 of Lemma 6.1, e must be of the form $e' :: [x{\leftarrow}c] :: e''$, so a $\to_{\mathtt{e}}$ transition applies. □

6.2 Right-to-Left Call-by-Value: the Leroy Abstract Machine

The transitions of the LAM are:

$$\begin{array}{c|c|c|c|c|c|c} \overline{t}\overline{u} & e & \pi & \to_{\mathtt{c}_1} & \overline{u} & e & \mathbf{f}(\overline{t},e)::\pi \\ \overline{v} & e & \mathbf{f}(\overline{t},e')::\pi & \to_{\mathtt{c}_2} & \overline{t} & e' & \mathbf{a}(\overline{v},e)::\pi \\ \lambda x.\overline{t} & e & \mathbf{a}(c)::\pi & \to_{\mathtt{m}} & \overline{t} & [x{\leftarrow}c]::e & \pi \\ x & e & \pi & \to_{\mathtt{e}} & \overline{t} & e' & \pi \end{array}$$

where $\to_{\mathtt{e}}$ takes place only if $e = e'' :: [x{\leftarrow}(\overline{t},e')] :: e'''$.

We omit all the proofs (that can be found in [9]) because they are minimal variations on those for the CEK.

Lemma 6.3 (LAM Invariants)**.** *Let $s = \overline{u} \mid e \mid \pi$ be a LAM reachable state whose initial code $\overline{t}$ is well-named. Then:*

1. Closure*: every closure in s is closed;*
2. Subterm*: any code in s is a literal subterm of $\overline{t}$;*
3. Value*: any code in e is a value and, for every element of π of the form $\mathbf{a}(\overline{u},e')$, $\overline{u}$ is a value;*
4. Contexts Decoding*: $\underline{\pi}$ and $\underline{\pi}\langle\underline{e}\rangle$ are right-to-left CBV evaluation contexts.*
5. Name*: any closure c in s is well-named and its names are names of $\overline{t}$ (i.e. $\Delta(c) \subseteq \mathtt{fv}(\overline{t})$).*
6. Environment Size*: the length of any environment in s is bound by $|\overline{t}|$.*

Theorem 6.4 (LAM Distillation)**.** *$(LAM, \mathtt{Value}^{\mathtt{RL}}, \equiv, \underline{\cdot})$ is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative 1*: if $s \to_{\mathtt{c}_1} s'$ then $\underline{s} \equiv \underline{s'}$;*
2. Commutative 2*: if $s \to_{\mathtt{c}_2} s'$ then $\underline{s} = \underline{s'}$.*
3. Multiplicative*: if $s \to_{\mathtt{m}} s'$ then $\underline{s} \multimap_{\mathtt{m}} \underline{s'}$;*
4. Exponential*: if $s \to_{\mathtt{e}} s'$ then $\underline{s} \multimap_{\mathtt{e}} \equiv \underline{s'}$;*

7. Towards Call-by-Need: the Split CEK and the MAM

In this section we study two further machines:

1. *The Split CEK (SCEK)*, obtained disentangling the two uses of the stack (for arguments and for functions) in the CEK. The split CEK can be seen as a simplification of Landin's SECD machine [35].
2. *The Milner Abstract Machine (MAM)*, that is a variation over the KAM with only one global environment and with just one global closure, what is sometimes called a *heap* or a *store*. Essentially, it unveils the content of distance rules at the machine level.

The ideas at work in these two case studies—both playing with the use of contexts—will be combined in the next section, obtaining a new simple CBNeed machine, the MAD.

7.1 The Split CEK, or Revisiting the SECD Machine

For the CEK machine we proved that the stack, that collects both arguments and functions, decodes to an evaluation context (Lemma 6.1.4). The new CBV machine in Fig. 2, deemed *Split CEK*, has two stacks: one for arguments and one for functions. Both will decode to evaluation contexts.

Note that the evaluation contexts V for the calculus $\mathtt{Value}^{\mathtt{LR}}$:

$$V ::= \langle\cdot\rangle \mid Vt \mid L\langle v\rangle V \mid V[x{\leftarrow}t]$$

have two cases for application. Essentially, when dealing with Vt the machine puts t in a stack for arguments (identical to the stack of the KAM), while in the case $L\langle v\rangle V$ the machine puts the closure (corresponding to) $L\langle v\rangle$ in a stack for functions, called *dump*.

Actually, together with the closure it also has to store the current argument stack, to not mess things up.

Thus, an entry of the function stack is a pair (c, π), where c is a closure $(\overline{v}, e)$, and the three components $\overline{v}$, e, and π together correspond to the evaluation context $\underline{\pi}\langle \underline{e}\langle \overline{v}\langle\cdot\rangle\rangle\rangle$.

Let us explain the idea at the level of the machine. Whenever the code is an abstraction $\overline{v}$ and the argument stack π is non-empty (*i.e.* $\pi = c :: \pi'$), the machine saves the active closure, given by current code $\overline{v}$ and environment e, and the tail of the stack π' by pushing a new entry $((\overline{v}, e), \pi')$ on the dump, and then starts evaluating the first closure c of the stack. The syntax for dumps then is

$$D ::= \epsilon \mid (c, \pi) :: D$$

Every dump decodes to a context according to:

$$\underline{\epsilon} \quad := \quad \langle\cdot\rangle \qquad\qquad \underline{((\overline{v}, e), \pi) :: D} \quad := \quad \underline{D}\langle\underline{\pi}\langle\underline{e}\langle\overline{v}\langle\cdot\rangle\rangle\rangle\rangle$$

Relationship with the SECD. For the acquainted reader, the new stack morally is the *dump* of Landin's SECD machine [35] (but beware that the original definition of the SECD is quite more technical). An in-depth analysis of the SECD machine can be found in Danvy's [22], where it is shown that the SECD implements right-to-left CBV, and not left-to-right CBV as the Split CEK. However, here we are rather interested in showing that *splitting the stack* is a general transformation, modularly captured by distillation and enlightening the dump of the SECD, which may look mysterious at first. It is enough to apply the same transformation to the LAM, getting a *Split LAM*, to get closer to the original SECD. Such an exercise, however, is left to the reader. In Sect. 9 we will instead apply the inverse transformation—merging two stacks into one—to a CBNeed machine.

Distillation. The decoding of terms, environments, closures, and stacks is as for the KAM. The decoding of states is defined as $\underline{\overline{t} \mid e \mid \pi \mid D} := \underline{D}\langle\underline{\pi}\langle\underline{e}\langle\overline{t}\rangle\rangle\rangle$. The proofs for the Split CEK are in [9].

Lemma 7.1 (Split CEK Invariants)**.** *Let $s = \overline{u} \mid e \mid \pi \mid D$ be a Split CEK reachable state whose initial code $\overline{t}$ is well-named. Then:*

1. Closure: *every closure in s is closed;*
2. Subterm: *any code in s is a literal subterm of $\overline{t}$;*
3. Value: *the code of any closure in the dump or in any environment in s is a value;*
4. Contextual Decoding*: $\underline{D}$, $\underline{D}\langle\underline{\pi}\rangle$, and $\underline{D}\langle\underline{\pi}\langle\underline{e}\rangle\rangle$ are left-to-right CBV evaluation context.*
5. Name*: any closure c in s is well-named and its names are names of $\overline{t}$ (i.e. $\Delta(c) \subseteq \mathtt{fv}(\overline{t})$).*
6. Environment Size*: the length of any environment in s is bound by $|\overline{t}|$.*

Theorem 7.2 (Split CEK Distillation)**.** $(\text{Split CEK}, \mathtt{Value}^{\mathrm{LR}}, \equiv, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative 1*: if $s \to_{\mathtt{c}_1} s'$ then $\underline{s} \equiv \underline{s'}$;*
2. Commutative 2*: if $s \to_{\mathtt{c}_2} s'$ then $\underline{s} \equiv \underline{s'}$;*
3. Multiplicative*: if $s \to_{\mathtt{m}} s'$ then $\underline{s} \multimap_{\mathtt{m}} \underline{s'}$;*
4. Exponential*: if $s \to_{\mathtt{e}} s'$ then $\underline{s} \multimap_{\mathtt{e}} \equiv \underline{s'}$.*

7.2 Milner Abstract Machine

The LSC suggests the design of a simpler version of the KAM, the *Milner Abstract Machine* (MAM), that avoids the concept of closure. At the language level, the idea is that, by repeatedly applying the axioms $\equiv_{dup}$ and $\equiv_{@}$ of the structural equivalence, explicit substitutions can be folded and brought *outside*. At the machine level, the local environments in the closures are replaced by just one global environment that closes the code and the stack, as well as the global environment itself.

Of course, naively turning to a global environment breaks the well-named invariant of the machine. This point is addressed using an α-renaming and name generation in the variable (or exponential) transition, *i.e.* when substitution takes place.

Here we employ the global environments E of Sect. 3 and we redefine stacks as $\pi ::= \epsilon \mid \overline{t} :: \pi$. A state of the MAM is given by a code $\overline{t}$, a stack π and a global environment E.

The transitions of the MAM are:

$$\begin{array}{c|c|c|c|c|c|c} \overline{t}\,\overline{u} & \pi & E & \to_{\mathtt{c}} & \overline{t} & \overline{u} :: \pi & E \\ \lambda x.\overline{t} & \overline{u} :: \pi & E & \to_{\mathtt{m}} & \overline{t} & \pi & [x \leftarrow \overline{u}] :: E \\ x & \pi & E & \to_{\mathtt{e}} & \overline{t}^{\alpha} & \pi & E \end{array}$$

where $\to_{\mathtt{e}}$ takes place only if $E = E''\langle E'[x \leftarrow \overline{t}]\rangle$ and $\overline{t}^{\alpha}$ is a well-named code α-equivalent to $\overline{t}$ and s.t. any bound name in $\overline{t}^{\alpha}$ is fresh with respect to those in π and E.

The decoding of a MAM state $\overline{t} \mid \pi \mid E$ is similar to the decoding of a KAM state, but the stack and the environment context are applied in reverse order (this is why stack and environment in MAM states are swapped with respect to KAM states):

$$\begin{array}{rcl@{\qquad}rcl} \underline{\epsilon} & := & \langle\cdot\rangle & \underline{[x \leftarrow \overline{t}] :: E} & := & \underline{E}\langle\langle\cdot\rangle[x \leftarrow \overline{t}]\rangle \\ \underline{\overline{t} :: \pi} & := & \underline{\pi}\langle\langle\cdot\rangle\overline{t}\rangle & \underline{\overline{t} \mid \pi \mid E} & := & \underline{E}\langle\underline{\pi}\langle\overline{t}\rangle\rangle \end{array}$$

To every MAM state $\overline{t} \mid \pi \mid E$ we associate the pair $(\underline{\pi}\langle\overline{t}\rangle, E)$ (note that $\underline{\pi}\langle\overline{t}\rangle$ now is a code, *i.e.* it does not contain explicit substitutions) and call it the *global closure* of the state.

As for the KAM, the decoding of contexts can be done statically, *i.e.* it does not need dynamic invariants.

Lemma 7.3 (Contextual Decoding)**.** *Let E be a global environment and π be a stack of the MAM. Then $\underline{E}$ is a substitution context, and both $\underline{\pi}$ and $\underline{\pi}\langle\underline{E}\rangle$ are evaluation contexts.*

For the dynamic invariants we need a different notion of closed closure.

Definition 7.4. *Given a global environment E and a code $\overline{t}$, we define by mutual induction two predicates E* is closed *and $(\overline{t}, E)$* is closed *as follows:*

$$\begin{array}{rcl} & & \epsilon \text{ is closed} \\ (\overline{t}, E) \text{ is closed} & \implies & [x \leftarrow t] :: E \text{ is closed} \\ \mathtt{fv}(\overline{t}) \subseteq \Delta(E) \wedge E \text{ is closed} & \implies & (\overline{t}, E) \text{ is closed} \end{array}$$

The dynamic invariants are:

Lemma 7.5 (MAM invariants)**.** *Let $s = \overline{u} \mid \pi \mid E$ be a MAM state reached by an execution ρ of initial well-named code $\overline{t}$. Then:*

1. Global Closure: *the global closure $(\underline{\pi}\langle\overline{t}\rangle, E)$ of s is closed;*
2. Subterm: *any code in s is a literal subterm of $\overline{t}$;*
3. Names: *the global closure of s is well-named;*
4. Environment Size: *the length of the global environment in s is bound by $|\rho|_m$.*

Abstract Considerations on Concrete Implementations. Note the new environment size invariant. The bound now depends on the length of the evaluation sequence ρ, not on the size of the initial term $\overline{t}$. If one implements $\to_{\mathtt{e}}$ looking for x in E sequentially, then each $\to_{\mathtt{e}}$ transition has cost $O(|\rho|_m)$, and the cost of implementing ρ is easily seen to become quadratic in $|\rho|$. Therefore—at first sight—the MAM is not a reasonable abstract machine (in the sense of Sect. 5). However, the MAM is meant to be implemented using a representation of codes pointers for variables, so that looking for x in E takes constant time. Then the global environment, even if formalized as a list, should rather be considered as a store.

$\overline{t}\overline{u}$	π	D	E	$\to_{\mathtt{c}_1}$	$\overline{t}$	$\overline{u} :: \pi$	D	E
$\lambda x.\overline{t}$	$\overline{u} :: \pi$	D	E	$\to_{\mathtt{m}}$	$\overline{t}$	π	D	$[x \leftarrow \overline{u}] :: E$
x	π	D	$E_1 :: [x \leftarrow \overline{t}] :: E_2$	$\to_{\mathtt{c}_2}$	$\overline{t}$	ϵ	$(E_1, x, \pi) :: D$	E_2
$\overline{v}$	ϵ	$(E_1, x, \pi) :: D$	E_2	$\to_{\mathtt{e}}$	$\overline{v}^{\alpha}$	π	D	$E_1 :: [x \leftarrow \overline{v}] :: E_2$

Figure 3. The Milner Abstract machine by-neeD (MAD).

The name invariant is what guarantees that variables can indeed be taken as pointers, as there is no name clash. Note that the cost of a $\to_{\mathtt{e}}$ transition is not constant, as the renaming operation actually makes $\to_{\mathtt{e}}$ linear in $|t|$ (by the subterm invariant). So, assuming a pointer-based representation, ρ can be implemented in time $O(|\rho| \cdot |\overline{t}|)$, as for local machines. In other words, the MAM is a reasonable abstract machine.

Theorem 7.6 (MAM Distillation)**.** $(\mathit{MAM}, \mathtt{Name}, \equiv, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state* s *we have:*

1. Commutative*: if* $s \to_{\mathtt{c}} s'$ *then* $\underline{s} = \underline{s'}$*;*
2. Multiplicative*: if* $s \to_{\mathtt{m}} s'$ *then* $\underline{s} \multimap_{\mathtt{m}} \equiv \underline{s'}$*;*
3. Exponential*: if* $s \to_{\mathtt{e}} s'$ *then* $\underline{s} \multimap_{\mathtt{e}} =_{\alpha} \underline{s'}$*.*

Proof. Properties of the decoding (*progress* is as for the KAM):

1. *Commutative*. In contrast to the KAM, $\to_{\mathtt{c}}$ gives a true identity:
$$\underline{\overline{t}\overline{u} \mid \pi \mid E} \quad = \quad \underline{E\langle \underline{\pi}\langle \overline{t}\overline{u}\rangle\rangle} \quad = \quad \underline{\overline{t} \mid \overline{u} :: \pi \mid E}$$
2. *Multiplicative*. Since substitutions and evaluation contexts commute via $\equiv$ (Lemma 2.4), $\to_{\mathtt{m}}$ maps to:
$$\begin{array}{rcll} \underline{\lambda x.\overline{t} \mid \overline{u} :: \pi \mid E} & = & \underline{E\langle \underline{\pi}\langle (\lambda x.\overline{t})\overline{u}\rangle\rangle} & \multimap_{\mathtt{m}} \\ & & \underline{E\langle \underline{\pi}\langle \overline{t}[x \leftarrow \overline{u}]\rangle\rangle} & \equiv_{Lem.2.4} \\ & & \underline{E\langle \underline{\pi}\langle \overline{t}\rangle[x \leftarrow \overline{u}]\rangle} & = \\ & & \underline{\overline{t} \mid \pi \mid [x \leftarrow \overline{u}] :: E} & \end{array}$$
3. *Exponential*. The erasure of part of the environment of the KAM is replaced by an explicit use of α-equivalence:
$$\begin{array}{rcll} \underline{x \mid \pi \mid E :: [x \leftarrow \overline{u}] :: E'} & = & \underline{E'\langle \underline{E}\langle \underline{\pi}\langle x\rangle\rangle[x \leftarrow \overline{u}]\rangle} & \multimap_{\mathtt{e}} \\ & & \underline{E'\langle \underline{E}\langle \underline{\pi}\langle \overline{u}\rangle\rangle[x \leftarrow \overline{u}]\rangle} & =_{\alpha} \\ & & \underline{E'\langle \underline{E}\langle \underline{\pi}\langle \overline{u}^{\alpha}\rangle\rangle[x \leftarrow \overline{u}]\rangle} & = \\ & & \underline{\overline{u}^{\alpha} \mid \pi \mid E :: [x \leftarrow \overline{u}] :: E'} & \end{array}$$

□

Digression about $\equiv$. Note that in the distillation theorem structural equivalence is used only to commute with stacks. The calculus and the machine in fact form a distillery also with respect to the following simpler notion of structural equivalence. Let $\equiv_{\mathrm{MAM}}$ be the smallest equivalence relation generated by the closure by (call-by-name) evaluation contexts of the axiom $\equiv_{@l}$ in Fig. 1 (page 5). The next lemma guarantees that $\equiv_{\mathrm{MAM}}$ is a strong bisimulation (the proof is in [9]), and so $\equiv_{\mathrm{MAM}}$ provides another MAM distillery.

Lemma 7.7. $\equiv_{\mathit{MAM}}$ *is a strong bisimulation with respect to* $\multimap$*.*

8. Call-by-Need: the MAD and the Merged MAD

In this section we introduce a new abstract machine for CBNeed, deemed *Milner Abstract machine by-neeD* (MAD). The MAD arises very naturally as a reformulation of the `Need` calculus of Sect. 2. The motivations behind the introduction of a new machine are:

1. *Simplicity*: the MAD is arguably simpler than all other CBNeed machines in the literature, in particular its distillation is very natural;
2. *Factorizing the Distillation of the Lazy KAM and of the SAM*: the study of the MAD will be followed by two sections showing how to tweak the MAD in order to obtain (simplifications of) two CBNeed machines in the literature, Cregut's Lazy KAM and Sestoft's machine (here called *SAM*). Expressing the Lazy KAM and the SAM as modifications of the MAD helps understanding their design, their distillation (that would otherwise look very technical), and their relationship;
3. *Simpler Reasoning*: for CBNeed the proof that distillation preserves complexity (in forthcoming Sect. 11) is subtle, and requires a global analysis. The MAD allows to reason on a simple machine. The reasoning is then easily seen to scale up to its modified versions, without having to deal from the start with their complex structure.
4. *Modularity of Our Contextual Theory of Abstract Machines*: the MAD is obtained by applying to the KAM the following two tweaks:
   (a) *Global Environments*: the MAD uses the global environment approach of the MAM to implement memoization;
   (b) *Dump*: the MAD uses the dump-like approach of the Split CEK/SECD to evaluate inside explicit substitutions;

8.1 The MAD

The MAD is shown in Fig. 3. Note that when the code is a variable the transition is now commutative. The idea is that whenever the code is a variable x and the environment has the form $E_1 :: [x \leftarrow \overline{t}] :: E_2$, the machine jumps to evaluate $\overline{t}$ saving the prefix of the environment E_1, the variable x on which it will substitute the result of evaluating $\overline{t}$, and the stack π. This is how hereditarily weak head evaluation context are implemented by the MAD.

Dumps (D) and their decoding are defined by
$$D ::= \epsilon \mid (E, x, \pi) :: D$$
$$\underline{\epsilon} \quad := \quad \langle\cdot\rangle \qquad \underline{(E, x, \pi) :: D} \quad := \quad \underline{E\langle \underline{D}\langle \underline{\pi}\langle x\rangle\rangle\rangle[x \leftarrow \langle\cdot\rangle]}$$

The decoding of terms, environments, and stacks is defined as for the KAM. The decoding of states is defined by $\underline{\overline{t} \mid \pi \mid D \mid E} := \underline{E\langle \underline{D}\langle \underline{\pi}\langle \overline{t}\rangle\rangle\rangle}$. The decoding of contexts is static:

Lemma 8.1 (Contextual Decoding)**.** *Let* D*,* π*, and* E *be a dump, a stack, and a global environment of the MAD, respectively. Then* $\underline{D}$*,* $\underline{D}\langle\underline{\pi}\rangle$*,* $\underline{E}\langle\underline{D}\rangle$*, and* $\underline{E}\langle\underline{D}\langle\underline{\pi}\rangle\rangle$ *are CBNeed evaluation contexts.*

Closed closures are defined as for the MAM. Given a state $s = \overline{t} \mid \pi \mid D \mid E_0$ with $D = (E_1, x_1, \pi_1) :: \ldots :: (E_n, x_n, \pi_n)$, its closures are $(\underline{\pi}\langle\overline{t}\rangle, E_0)$ and, for $i \in \{1, \ldots, n\}$,
$$(\underline{\pi_i}\langle x_i\rangle, E_i :: [x_i \leftarrow \underline{\pi_{i-1}}\langle x_{i-1}\rangle] :: \ldots :: [x_1 \leftarrow \underline{\pi}\langle\overline{t}\rangle] :: E_0).$$

The dynamic invariants are:

Lemma 8.2 (MAD invariants)**.** *Let* $s = \overline{t} \mid \pi \mid D \mid E_0$ *be a MAD reachable state whose initial code* $\overline{t}$ *is well-named, and s.t.* $D = (E_1, x_1, \pi_1) :: \ldots :: (E_n, x_n, \pi_n)$*. Then:*

1. Global Closure: *the closures of* s *are closed;*
2. Subterm: *any code in* s *is a literal subterm of* $\overline{t}$*;*
3. Names: *the closures of* s *are well-named.*

$$\begin{array}{c|c|ccc|c|c}
\overline{t}\,\overline{u} & \pi & E & \to_{\mathtt{c}_1} & \overline{t} & \mathbf{a}(\overline{u}) :: \pi & E \\
\lambda x.\overline{t} & \mathbf{a}(\overline{u}) :: \pi & E & \to_{\mathtt{m}} & \overline{t} & \pi & [x \leftarrow \overline{u}] :: E \\
x & \pi & E_1 :: [x \leftarrow \overline{t}] :: E_2 & \to_{\mathtt{c}_2} & \overline{t} & \mathbf{h}(E_1, x) :: \pi & E_2 \\
\overline{v} & \mathbf{h}(E_1, x) :: \pi & E_2 & \to_{\mathtt{e}} & \overline{v}^{\alpha} & \pi & E_1 :: [x \leftarrow \overline{v}] :: E_2
\end{array}$$

Figure 4. The Merged MAD.

For the properties of the decoding function please note that, as defined in Sect. 2, the structural congruence $\equiv_{\mathtt{Need}}$ for CBNeed is different from before.

Theorem 8.3 (MAD Distillation)**.** $(\mathrm{MAD}, \mathtt{Need}, \equiv_{\mathtt{Need}}, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative 1*: if $s \to_{\mathtt{c}_1} s'$ then $\underline{s} = \underline{s'}$;*
2. Commutative 2*: if $s \to_{\mathtt{c}_2} s'$ then $\underline{s} = \underline{s'}$;*
3. Multiplicative*: if $s \to_{\mathtt{m}} s'$ then $\underline{s} \multimap_{\mathtt{m}} \equiv_{\mathtt{Need}} \underline{s'}$;*
4. Exponential*: if $s \to_{\mathtt{e}} s'$ then $\underline{s} \multimap_{\mathtt{e}} =_{\alpha} \underline{s'}$.*

Proof. 1. *Commutative 1*.

$$\underline{\overline{t}\,\overline{u} \mid \pi \mid D \mid E} = \underline{E}\langle \underline{D}\langle \underline{\pi}\langle \overline{t}\,\overline{u}\rangle\rangle\rangle = \underline{\overline{t} \mid \overline{u} :: \pi \mid D \mid E}$$

2. *Commutative 2*:

$$\begin{array}{rcl}
\underline{x \mid \pi \mid D \mid E_1 :: [x \leftarrow \overline{t}] :: E_2} & = & \underline{E_2}\langle \underline{E_1}\langle \underline{D}\langle \underline{\pi}\langle x\rangle\rangle\rangle[x \leftarrow \overline{t}]\rangle \\
& = & \underline{\overline{t} \mid \epsilon \mid (E_1, x, \pi) :: D \mid E_2}
\end{array}$$

3. *Multiplicative*.

$$\begin{array}{rcll}
\underline{\lambda x.\overline{t} \mid \overline{u} :: \pi \mid D \mid E} & = & \underline{E}\langle \underline{D}\langle \underline{\pi}\langle (\lambda x.\overline{t})\,\overline{u}\rangle\rangle\rangle & \multimap_{\mathtt{m}} \\
& & \underline{E}\langle \underline{D}\langle \underline{\pi}\langle \overline{t}[x \leftarrow \overline{u}]\rangle\rangle\rangle & \equiv_{\mathtt{Need}} \text{Lem. 2.4} \\
& & \underline{E}\langle \underline{D}\langle \underline{\pi}\langle \overline{t}\rangle\rangle[x \leftarrow \overline{u}]\rangle & = \\
& & \underline{\overline{t} \mid \pi \mid D \mid [x \leftarrow \overline{u}] :: E} &
\end{array}$$

Note that to apply Lemma 2.4 we use the global closure invariant, as $\overline{u}$, being on the stack, is closed by E and so $\underline{D}$ does not capture its free variables.

4. *Exponential*.

$$\begin{array}{rcl}
\underline{\overline{v} \mid \epsilon \mid (E_1, x, \pi) :: D \mid E_2} & = & \underline{E_2}\langle \underline{E_1}\langle \underline{D}\langle \underline{\pi}\langle x\rangle\rangle\rangle[x \leftarrow \overline{v}]\rangle \\
& \multimap_{\mathtt{e}} & \underline{E_2}\langle \underline{E_1}\langle \underline{D}\langle \underline{\pi}\langle \overline{v}\rangle\rangle\rangle[x \leftarrow \overline{v}]\rangle \\
& =_{\alpha} & \underline{E_2}\langle \underline{E_1}\langle \underline{D}\langle \underline{\pi}\langle \overline{v}^{\alpha}\rangle\rangle\rangle[x \leftarrow \overline{v}]\rangle \\
& = & \underline{\overline{v}^{\alpha} \mid \pi \mid D \mid E_1 :: [x \leftarrow \overline{v}] :: E_2}
\end{array}$$

Progress. Let $s = \overline{t} \mid \pi \mid D \mid E$ be a commutative normal form s.t. $\underline{s} \multimap u$. If $\overline{t}$ is

1. *an application $\overline{u}\overline{w}$*. Then a $\to_{\mathtt{c}_1}$ transition applies and s is not a commutative normal form, absurd;
2. *an abstraction v*. The decoding $\underline{s}$ is of the form $\underline{E}\langle \underline{D}\langle \underline{\pi}\langle v\rangle\rangle\rangle$. The stack π and the dump D cannot both be empty, since then $\underline{s} = \underline{E}\langle v\rangle$ would be normal. So either the stack is empty and a $\to_{\mathtt{e}}$ transition applies, or the stack is not empty and a $\to_{\mathtt{m}}$ transition applies;
3. *a variable x*. By Lemma 8.2.1 it must be bound by E, so a $\to_{\mathtt{c}_2}$ transition applies, and s is not a commutative normal form, absurd. □

Abstract Considerations on Concrete Implementations. Consider transition $\to_{\mathtt{c}_2}$. Note that the saving of the prefix E_1 in the dump forces to have E implemented as a list, and so to go through E sequentially. This fact goes against the intuition that E is a store (rather than a list), and makes the MAD an unreasonable abstract machine (see the analogous considerations for the KAM and for the MAM). To solve this point, in Sect. 10 we will present the Pointing MAD, a variant of the MAD (akin to Sestoft's machine for CBNeed [44]) that avoids saving E_1 in a dump entry, and restoring the store view of the global environment. The detour is justified as follows:

1. the Pointing MAD is more involved;
2. for the complexity analysis of distillation in Sect. 11 it is easier to reason on the MAD;
3. this issue about concrete implementations is orthogonal to the complexity analysis of the distillation process.

9. The Merged MAD, or Revisiting the Lazy KAM

Splitting the stack of the CEK machine in two we obtained a simpler form of the SECD machine. In this section we apply to the MAD the reverse transformation. The result is a machine, deemed *Merged MAD*, having only one stack and that can be seen as a simpler version of Cregut's lazy KAM [19] (but we are rather inspired by Danvy and Zerny's presentation in [24]).

To distinguish the two kinds of objects on the stack we use a marker, as for the CEK and the LAM. Formally, the syntax for stacks is:

$$\pi ::= \epsilon \mid \mathbf{a}(\overline{t}) :: \pi \mid \mathbf{h}(E, x) :: \pi$$

where $\mathbf{a}(\overline{t})$ denotes a term to be used as an argument (as for the CEK) and $\mathbf{h}(E, x, \pi)$ is morally an entry of the dump of the MAD, where however there is no need to save the current stack. The transitions of the Merged MAD are in Fig. 4.

The decoding is defined as follows

$$\begin{array}{rcl}
\underline{\epsilon} & := & \langle\cdot\rangle \\
\underline{[x \leftarrow \overline{t}] :: E} & := & \underline{E}\langle\langle\cdot\rangle[x \leftarrow \overline{t}]\rangle \\
\underline{\mathbf{h}(E, x) :: \pi} & := & \underline{E}\langle \underline{\pi}\langle x\rangle\rangle[x \leftarrow \langle\cdot\rangle] \\
\underline{\mathbf{a}(\overline{t}) :: \pi} & := & \underline{\pi}\langle\langle\cdot\rangle\overline{t}\rangle \\
\underline{\overline{t} \mid \pi \mid E} & := & \underline{E}\langle \underline{\pi}\langle\overline{t}\rangle\rangle
\end{array}$$

Lemma 9.1 (Contextual Decoding)**.** *Let π and E be a stack and a global environment of the Merged MAD. Then $\underline{\pi}$ and $\underline{E}\langle\underline{\pi}\rangle$ are CBNeed evaluation contexts.*

The dynamic invariants of the Merged MAD are exactly the same of the MAD, with respect to an analogous set of closures associated to a state (whose exact definition is omitted). The proof of the following theorem—almost identical to that of the MAD—is in [9].

Theorem 9.2 (Merged MAD Distillation)**.** $(\text{Merged MAD}, \mathtt{Need}, \equiv_{\mathtt{Need}}, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative 1*: if $s \to_{\mathtt{c}_1} s'$ then $\underline{s} = \underline{s'}$;*
2. Commutative 2*: if $s \to_{\mathtt{c}_2} s'$ then $\underline{s} = \underline{s'}$;*
3. Multiplicative*: if $s \to_{\mathtt{m}} s'$ then $\underline{s} \multimap_{\mathtt{m}} \equiv_{\mathtt{Need}} \underline{s'}$;*
4. Exponential*: if $s \to_{\mathtt{e}} s'$ then $\underline{s} \multimap_{\mathtt{e}} =_{\alpha} \underline{s'}$.*

$\overline{t}\,\overline{u}$	π	D	E	$\to_{\mathtt{c}_1}$	$\overline{t}$	$\overline{u} :: \pi$	D	E
$\lambda x.\overline{t}$	$\overline{u} :: \pi$	ϵ	E	$\to_{\mathtt{m}_1}$	$\overline{t}$	π	ϵ	$[x \leftarrow \overline{u}] :: E$
$\lambda x.\overline{t}$	$\overline{u} :: \pi$	$(y, \pi') :: D$	$E_1 :: [y \leftarrow \Box] :: E_2$	$\to_{\mathtt{m}_2}$	$\overline{t}$	π	$(y, \pi') :: D$	$E_1 :: [y \leftarrow \Box] :: [x \leftarrow \overline{u}] :: E_2$
x	π	D	$E_1 :: [x \leftarrow \overline{t}] :: E_2$	$\to_{\mathtt{c}_2}$	$\overline{t}$	ϵ	$(x, \pi) :: D$	$E_1 :: [x \leftarrow \Box] :: E_2$
$\overline{v}$	ϵ	$(x, \pi) :: D$	$E_1 :: [x \leftarrow \Box] :: E_2$	$\to_{\mathtt{e}}$	$\overline{v}^{\alpha}$	π	D	$E_1 :: [x \leftarrow \overline{v}] :: E_2$

Figure 5. The Pointing MAD.

10. The Pointing MAD, or Revisiting the SAM

In the MAD, the global environment is divided between the environment of the machine and the entries of the dump. On the one hand, this choice makes the decoding very natural. On the other hand, one would like to keep the global environment in just one place, to validate the intuition that it is a store rather than a list, and let the dump only collect variables and stacks. This is what we do here, exploiting the fact that variable names can be taken as pointers (see the *abstract considerations* in Sect. 7.2 and Sect. 8.1).

The new machine, called Pointing MAD, is in Fig. 5, and uses a new dummy constant $\Box$ for the substitutions whose variable is in the dump. It also has two multiplicative transitions, that will both distilled into $\multimap_{\mathtt{m}}$, depending on the content of the dump. It can be seen as a simpler version of *Sestoft's Abstract Machine* [44], here called SAM. Dumps and environments are defined by:

$$\begin{array}{rcl} D & ::= & \epsilon \mid (x, \pi) :: D \\ E & ::= & \epsilon \mid [x \leftarrow \overline{t}] :: E \mid [x \leftarrow \Box] :: E \end{array}$$

A substitution of the form $[x \leftarrow \Box]$ is *dumped*, and we also say that x is dumped.

Note that the variables of the entries in D appear in reverse order with respect to the corresponding substitutions in E. We will show that fact is an invariant, called *duality*.

Definition 10.1 (Duality $E \bot D$)**.** *Duality $E \bot D$ between environments and dumps is defined by*

1. $\epsilon \bot \epsilon$*;*
2. $E :: [x \leftarrow \overline{t}] \bot D$ *if* $E \bot D$*;*
3. $E :: [x \leftarrow \Box] \bot (x, \pi) :: D$ *if* $E \bot D$*.*

Note that in a dual pair the environment is always at least as long as the dump. A dual pair $E \bot D$ decodes to a context as follows:

$$\begin{array}{rcl} \underline{(E, \epsilon)} & := & \underline{E} \\ \underline{(E :: [x \leftarrow \Box], (x, \pi) :: D)} & := & \underline{(E, D)\langle \underline{\pi}\langle x \rangle\rangle[x \leftarrow \langle\cdot\rangle]} \\ \underline{(E :: [x \leftarrow \overline{t}], (y, \pi) :: D)} & := & \underline{(E, (y, \pi) :: D)[x \leftarrow \overline{t}]} \end{array}$$

The analysis of the Pointing MAD is based on a complex invariant that includes duality plus a generalization of the global closure invariant. We need an auxiliary definition:

Definition 10.2. *Given an environment E, we define its* slice $E\!\uparrow$ *as the sequence of substitutions after the rightmost dumped substitution. Formally:*

$$\begin{array}{rcl} \epsilon\!\uparrow & := & \epsilon \\ (E :: [x \leftarrow \overline{t}])\!\uparrow & := & E\!\uparrow :: [x \leftarrow \overline{t}] \\ (E :: [x \leftarrow \Box])\!\uparrow & := & \epsilon \end{array}$$

Moreover, if an environment E is of the form $E_1 :: [x \leftarrow \Box] :: E_2$, we define $E\!\uparrow_x := E_1\!\uparrow :: [x \leftarrow \Box] :: E_2$.

The notion of closed closure with global environment (Sect. 7.2) is extended to dummy constants $\Box$ as expected.

Lemma 10.3 (Pointing MAD invariants)**.** *Let $s = \overline{t} \mid E \mid \pi \mid D$ be a Pointing MAD reachable state whose initial code $\overline{t}$ is well-named. Then:*

1. Subterm: *any code in s is a literal subterm of $\overline{t}$;*
2. Names: *the global closure of s is well-named.*
3. Dump-Environment Duality:
   (a) *$(\underline{\pi}\langle\overline{t}\rangle, E\!\uparrow)$ is closed;*
   (b) *for every pair (x, π') in D, $(\underline{\pi'}\langle x\rangle, E\!\uparrow_x)$ is closed;*
   (c) *$E \bot D$ holds.*
4. Contextual Decoding: *$\underline{(E, D)}$ is a CBNeed evaluation context.*

Proof. See [9]. □

The decoding of a state is defined as $\underline{\overline{t} \mid \pi \mid D \mid E} := \underline{(E, D)}\langle\underline{\pi}\langle\overline{t}\rangle\rangle$.

Theorem 10.4 (Pointing MADDistillation)**.** *$(\text{Pointing MAD}, \mathtt{Need}, \equiv_{\mathtt{Need}}, \underline{\cdot})$ is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative 1 & 2*: if $s \to_{\mathtt{c}_1} s'$ or $s \to_{\mathtt{c}_2} s'$ then $\underline{s} = \underline{s'}$;*
2. Multiplicative 1 & 2*: if $s \to_{\mathtt{m}_1} s'$ or $s \to_{\mathtt{m}_2} s'$ then $\underline{s} \multimap_{\mathtt{m}} \equiv_{\mathtt{Need}} \underline{s'}$;*
3. Exponential*: if $s \to_{\mathtt{e}} s'$ then $\underline{s} \multimap_{\mathtt{e}=\alpha} \underline{s'}$;*

Proof. Properties of the decoding:

1. *Commutative 1*. We have
$$\underline{\overline{t}\,\overline{u} \mid \pi \mid D \mid E} \;=\; \underline{(E, D)\langle\underline{\pi}\langle\overline{t}\,\overline{u}\rangle\rangle} \;=\; \underline{\overline{t} \mid \overline{u} :: \pi \mid D \mid E}$$
2. *Commutative 2*. Note that E_2 has no dumped substitutions, since $E_1 :: [x \leftarrow \Box] :: E_2 \bot (x, \pi) :: D$. Then:
$$\begin{array}{ll} \underline{x \mid \pi \mid D \mid E_1 :: [x \leftarrow \overline{t}] :: E_2} & = \\ \underline{E_2}\langle\underline{(E_1, D)\langle\underline{\pi}\langle x\rangle\rangle[x \leftarrow \overline{t}]}\rangle & = \\ \underline{\overline{t} \mid \epsilon \mid (x, \pi) :: D \mid E_1 :: [x \leftarrow \Box] :: E_2} & \end{array}$$
3. *Multiplicative 1, empty dump*.
$$\begin{array}{lll} \underline{\lambda x.\overline{t} \mid \overline{u} :: \pi \mid \epsilon \mid E} = \underline{E}\langle\underline{\pi}\langle(\lambda x.\overline{t})\,\overline{u}\rangle\rangle & \multimap_{\mathtt{m}} \\ \underline{E}\langle\underline{\pi}\langle\overline{t}[x \leftarrow \overline{u}]\rangle\rangle & \equiv^{*}_{@l} \text{ Lem. 2.4} \\ \underline{E}\langle\underline{\pi}\langle\overline{t}\rangle\rangle[x \leftarrow \overline{u}] & = \\ \underline{\overline{t} \mid \pi \mid \epsilon \mid [x \leftarrow \overline{u}] :: E} & \end{array}$$
4. *Multiplicative 2, non-empty dump*.
$$\begin{array}{ll} \underline{\lambda x.\overline{t} \mid \overline{u} :: \pi \mid (y, \pi') :: D \mid E_1 :: [y \leftarrow \Box] :: E_2} & = \\ \underline{E_2}\langle\underline{(E_1, D)\langle\underline{\pi'}\langle y\rangle\rangle[y \leftarrow \underline{\pi}\langle(\lambda x.\overline{t})\,\overline{u}\rangle]}\rangle & \multimap_{\mathtt{m}} \\ \underline{E_2}\langle\underline{(E_1, D)\langle\underline{\pi'}\langle y\rangle\rangle[y \leftarrow \underline{\pi}\langle\overline{t}[x \leftarrow \overline{u}]\rangle]}\rangle & \equiv_{\mathtt{Need}} \text{ Lem. 2.4} \\ \underline{E_2}\langle\underline{(E_1, D)\langle\underline{\pi'}\langle y\rangle\rangle[y \leftarrow \underline{\pi}\langle\overline{t}\rangle][x \leftarrow \overline{u}]}\rangle & = \\ \underline{\overline{t} \mid \pi \mid (y, \pi') :: D \mid E_1 :: [y \leftarrow \Box] :: [x \leftarrow \overline{u}] :: E_2} & \end{array}$$

5. *Exponential*.

$$\begin{array}{ll} \overline{v} \mid \epsilon \mid (x,\pi) :: D \mid E_1 :: [x \leftarrow \Box] :: E_2 & = \\ \underline{E_2}\langle (E_1, D)\langle \underline{\pi}\langle x \rangle\rangle[x \leftarrow v]\rangle & \multimap_{\mathtt{e}} \\ \underline{E_2}\langle (E_1, D)\langle \underline{\pi}\langle v \rangle\rangle[x \leftarrow v]\rangle & =_{\alpha} \\ \underline{E_2}\langle (E_1, D)\langle \underline{\pi}\langle v^{\alpha} \rangle\rangle[x \leftarrow v]\rangle & = \\ \underline{\overline{v}^{\alpha} \mid \pi \mid D \mid E_1 :: [x \leftarrow \overline{v}] :: E_2} & \end{array}$$

Progress. Let $s = \bar{t} \mid \pi \mid D \mid E$ be a commutative normal form s.t. $\underline{s} \multimap u$. If $\bar{t}$ is

- *an application* $\overline{uw}$. Then a $\rightarrow_{\mathtt{c}_1}$ transition applies and s is not a commutative normal form, absurd.
- *a variable* x. By the machine invariant, x must be bound by $E\downarrow$. So $E = E_1 :: [x \leftarrow \overline{u}] :: E_2$, a $\rightarrow_{\mathtt{c}_2}$ transition applies, and s is not a commutative normal form, absurd.
- *an abstraction* $\overline{v}$. Two cases:
  - *The stack π is empty*. The dump D cannot be empty, since if $D = \epsilon$ we have that $\underline{s} = \underline{e}\langle \overline{v} \rangle$ is normal. So $D = (x, \pi') :: D'$. By duality, $E = E_1 :: [x \leftarrow \Box] :: E_2$ and a $\rightarrow_{\mathtt{e}}$ transition applies;
  - *The stack π is non-empty*. If the dump D is empty, the first case of $\rightarrow_{\mathtt{m}}$ applies. If $D = (x, \pi') :: D'$, by duality $E = E_1 :: [x \leftarrow \Box] :: E_2$ and the second case of $\rightarrow_{\mathtt{m}}$ applies. □

11. Distillation Preserves Complexity

Here, for every abstract machine we bound the number of commutative steps $|\rho|_{\mathtt{c}}$ in an execution ρ in terms of

1. the number of principal steps $|\rho|_p$,
2. the size $|\bar{t}|$ of the initial code $\bar{t}$.

The analysis only concerns the machines, but via the distillation theorems it expresses the length of the machine executions as a linear function of the length of the distilled derivations in the calculi. For every distillery, we will prove that the relationship is linear in both parameters, namely $|\rho|_{\mathtt{c}} = O((|\bar{t}| + 1) \cdot |\rho|_p)$ holds.

Definition 11.1. *Let* $\mathtt{M}$ *be a distilled abstract machine and* $\rho : s \rightarrow^* s'$ *be an execution of initial code* $\bar{t}$. $\mathtt{M}$ *is*

1. ***Globally bilinear*** *if* $|\rho|_{\mathtt{c}} = O((|\bar{t}| + 1) \cdot |\rho|_p)$.
2. ***Locally linear*** *if whenever* $s' \rightarrow_{\mathtt{c}}^{k} s''$ *then* $k = O(|\bar{t}|)$.

The next lemma shows that local linearity is a sufficient condition for global bilinearity.

Proposition 11.2 (Locally Linear ⇒ Globally Bilinear)**.** *Let* $\mathtt{M}$ *be a locally linear distilled abstract machine, and* ρ *an execution of initial code* $\bar{t}$. *Then* $\mathtt{M}$ *is globally bilinear.*

Proof. The execution ρ writes uniquely as $\rightarrow_{\mathtt{c}}^{k_1} \rightarrow_p^{h_1} \ldots \rightarrow_{\mathtt{c}}^{k_m} \rightarrow_p^{h_m}$. By hypothesis $k_i = O(|\bar{t}|)$ for every $i \in \{1, \ldots, m\}$. From $m \leq |\rho|_p$ follows that $|\rho|_{\mathtt{c}} = O(|\bar{t}| \cdot |\rho|_p)$. We conclude with $|\rho| = |\rho|_p + |\rho|_{\mathtt{c}} = |\rho|_p + O(|\bar{t}| \cdot |\rho|_p) = O((|\bar{t}| + 1) \cdot |\rho|_p)$. □

CBN and CBV machines are easily seen to be locally linear, and thus globally bilinear.

Theorem 11.3. *KAM, MAM, CEK, LAM, and the Split CEK are locally linear, and so also globally bilinear.*

Proof. 1. *KAM/MAM*. Immediate: $\rightarrow_{\mathtt{c}}$ reduces the size of the code, that is bounded by $|\bar{t}|$ by the subterm invariant.
2. *CEK*. Consider the following measure for states:

$$\#(\overline{u} \mid e \mid \pi) := \begin{cases} |\overline{u}| + |\overline{w}| & \text{if } \pi = \mathbf{a}(\overline{w}, e') :: \pi' \\ |\overline{u}| & \text{otherwise} \end{cases}$$

By direct inspection of the rules, it can be seen that both $\rightarrow_{\mathtt{c}_1}$ and $\rightarrow_{\mathtt{c}_2}$ transitions decrease the value of $\#$ for CEK states, and so the relation $\rightarrow_{\mathtt{c}_1} \cup \rightarrow_{\mathtt{c}_2}$ terminates (on reachable states). Moreover, both $|\overline{u}|$ and $|\overline{w}|$ are bounded by $|\bar{t}|$ by the subterm invariant (Lemma 6.1.2), and so $k \leq 2 \cdot |\bar{t}| = O(|\bar{t}|)$.
3. *LAM* and *Split CEK*. Similar to Point 2, see [9]. □

CBNeed machines are not locally linear, because a sequence of $\rightarrow_{\mathtt{c}_2}$ steps (remember $\rightarrow_{\mathtt{c}} := \rightarrow_{\mathtt{c}_1} \cup \rightarrow_{\mathtt{c}_2}$) can be as long as the global environment E, that is not bound by $|\bar{t}|$ but only by the number $|\rho|_p$ of preceding principal transitions (as for the MAM). Adapting the previous reasoning to this other bound would only show that globally $|\rho|_{\mathtt{c}}$ is quadratic in $|\rho|_p$, not linear. Luckily, being locally linear is not a necessary condition for global bilinearity. We are in fact going to show that CBNeed machines are globally bilinear. The key observation is that $|\rho|_{c_2}$ is not only locally but also globally bound by $|\rho|_p$, as the next lemma formalizes.

We treat the MAD. The reasoning for the Merged/Pointing MAD is analogous. Define $|\epsilon| := 0$ and $|(E, x, \pi) :: D| := 1 + |D|$.

Lemma 11.4. *Let* $s = \bar{t} \mid \pi \mid D \mid E$ *be a MAD state, reached by the execution* ρ. *Then*

1. $|\rho|_{c_2} = |\rho|_e + |D|$.
2. $|E| + |D| \leq |\rho|_m$
3. $|\rho|_{c_2} \leq |\rho|_e + |\rho|_m = |\rho|_p$

Proof. 1. Immediate, as $\rightarrow_{\mathtt{c}_2}$ is the only transition that pushes elements on D and $\rightarrow_{\mathtt{e}}$ is the only transition that pops them.
2. The only rule that produces substitutions is $\rightarrow_{\mathtt{m}}$. Note that 1) $\rightarrow_{\mathtt{c}_2}$ and $\rightarrow_{\mathtt{e}}$ preserve the global number of substitutions in a state; 2) E and D are made out of substitutions, if one considers every entry (E, x, π) of the dump as a substitution on x (and so the statement follows); 3) the inequality is given by the fact that an entry of the dump stocks an environment (counting for many substitutions).
3. Substitute Point 2 in Point 1. □

Theorem 11.5. *The MAD has globally linear commutations.*

Proof. Let ρ be an execution of initial code $\bar{t}$. Define $\rightarrow_{\neg c_1} := \rightarrow_{\mathtt{e}} \cup \rightarrow_{\mathtt{m}} \cup \rightarrow_{\mathtt{c}_2}$ and note $|\rho|_{\neg c_1}$ the number of its steps in ρ. We estimate $\rightarrow_{\mathtt{c}} := \rightarrow_{\mathtt{c}_1} \cup \rightarrow_{\mathtt{c}_2}$ by studying its components separately. For $\rightarrow_{\mathtt{c}_2}$, Lemma 11.4.3 proves $|\rho|_{c_2} \leq |\rho|_p = O(|\rho|_p)$. For $\rightarrow_{\mathtt{c}_1}$, as for the KAM, the length of a maximal $\rightarrow_{\mathtt{c}_1}$ subsequence of ρ is bounded by $|\bar{t}|$. The number of $\rightarrow_{\mathtt{c}_1}$ maximal subsequences of ρ is bounded by $|\rho|_{\neg c_1}$, that by Lemma 11.4.3 is linear in $O(|\rho|_p)$. Then $|\rho|_{c_1} = O(|\bar{t}| \cdot |\rho|_p)$. Summing up,

$$|\rho|_{c_2} + |\rho|_{c_1} = O(|\rho|_p) + O(|\bar{t}| \cdot |\rho|_p) = O((|\bar{t}| + 1) \cdot |\rho|_p)$$ □

The analysis presented here is complemented by the study in [8], where the number of exponential steps $\multimap_{\mathtt{e}}$ in a derivation d is shown to be polynomial (actually quadratic in CBN and linear in CBV and CBNeed) in terms of the number of multiplicative steps $\multimap_{\mathtt{m}}$ in d. Given our distillation theorems, the results in [8] equivalently relate the exponential and multiplicative transitions of the abstract machines. This derived analysis of principal transitions is a fruitful by-product of distilling abstract machines in the LSC.

12. Conclusions

The novelty of our study is the use of the linear substitution calculus (LSC) to discriminate between abstract machine transitions: some of them—the principal ones—are simulated, and thus shown to be logically relevant, while the others—the commutative ones—are mapped to the structural congruence and have to be considered as bookkeeping operations. On one hand, the LSC is a sharp tool to

study abstract machines. On the other hand, it provides an alternative to abstract machines which is *simpler* while being *conservative* at the level of complexity analysis.

Acknowledgments

A special acknowledgment to Claudio Sacerdoti Coen, for many useful discussions, comments and corrections to the paper. In particular, we owe him the intuition that a global analysis of call-by-need commutative rules may provide a linear bound. This work was partially supported by the ANR projects LOGOI (10-BLAN-0213-02) and COQUAS (ANR-12-JS02-006-01), by the French-Argentinian Laboratory in Computer Science **INFINIS**, the French-Argentinian project **ECOS-Sud** A12E04, the Qatar National Research Fund under grant NPRP 09-1107-1-168.

References

[1] B. Accattoli. An abstract factorization theorem for explicit substitutions. In *RTA*, pages 6–21, 2012.

[2] B. Accattoli. Linear logic and strong normalization. In *RTA*, pages 39–54, 2013.

[3] B. Accattoli. Evaluating functions as processes. In *TERMGRAPH*, pages 41–55, 2013.

[4] B. Accattoli and U. Dal Lago. On the invariance of the unitary cost model for head reduction. In *RTA*, pages 22–37, 2012.

[5] B. Accattoli and U. Dal Lago. Beta Reduction is Invariant, Indeed. Accepted to LICS/CSL 2014, 2014.

[6] B. Accattoli and D. Kesner. The structural λ-calculus. In *CSL*, pages 381–395, 2010.

[7] B. Accattoli and L. Paolini. Call-by-value solvability, revisited. In *FLOPS*, pages 4–16, 2012.

[8] B. Accattoli and C. Sacerdoti Coen. On the Value of Variables. Accepted to WOLLIC 2014, 2014.

[9] B. Accattoli, P. Barenbaum, and D. Mazza. Distilling Abstract Ma chines (Long Version). Available at `http://arxiv.org/abs/1406.2370`, 2014.

[10] B. Accattoli, E. Bonelli, D. Kesner, and C. Lombardi. A nonstandard standardization theorem. In *POPL*, pages 659–670, 2014.

[11] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, pages 8–19, 2003.

[12] M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inf. Process. Lett.*, 90(5):223–232, 2004.

[13] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997.

[14] Z. M. Ariola, A. Bohannon, and A. Sabry. Sequent calculi and abstract machines. *ACM Trans. Program. Lang. Syst.*, 31(4), 2009.

[15] M. Biernacka and O. Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theor. Comput. Sci.*, 375(1-3):76–108, 2007.

[16] M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Log.*, 9(1), 2007.

[17] S. Chang and M. Felleisen. The call-by-need lambda calculus, revisited. In *ESOP*, pages 128–147, 2012.

[18] P. Clairambault. Estimation of the length of interactions in arena game semantics. In *FOSSACS*, pages 335–349, 2011.

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

[20] V. Danos and L. Regnier. Head linear reduction. Technical report, 2004.

[21] V. Danos, H. Herbelin, and L. Regnier. Game semantics & abstract machines. In *LICS*, pages 394–405, 1996.

[22] O. Danvy. A rational deconstruction of landin's secd machine. In *IFL*, pages 52–71, 2004.

[23] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Technical Report RS-04-26, BRICS, 2004.

[24] O. Danvy and I. Zerny. A synthetic operational account of call-by-need evaluation. In *PPDP*, pages 97–108, 2013.

[25] N. G. de Bruijn. Generalizing Automath by Means of a Lambda-Typed Lambda Calculus. In *Mathematical Logic and Theoretical Computer Science*, number 106 in Lecture Notes in Pure and Applied Mathematics, pages 71–92. Marcel Dekker, 1987.

[26] R. Di Cosmo, D. Kesner, and E. Polonovski. Proof nets and explicit substitutions. *Math. Str. in Comput. Sci.*, 13(3):409–450, 2003.

[27] T. Ehrhard and L. Regnier. Böhm trees, Krivine's machine and the Taylor expansion of lambda-terms. In *CiE*, pages 186–197, 2006.

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

[29] R. Garcia, A. Lumsdaine, and A. Sabry. Lazy evaluation and delimited control. In *POPL*, pages 153–164, 2009.

[30] T. Hardin and L. Maranget. Functional runtime systems within the lambda-sigma calculus. *J. Funct. Program.*, 8(2):131–176, 1998.

[31] D. Kesner. A theory of explicit substitutions with safe and full composition. *Logical Methods in Computer Science*, 5(3), 2009.

[32] D. Kesner and S. Lengrand. Resource operators for lambda-calculus. *Inf. Comput.*, 205(4):419–473, 2007.

[33] D. Kesner and F. Renaud. The prismoid of resources. In *MFCS*, pages 464–476, 2009.

[34] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[35] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, Jan. 1964. . URL `http://dx.doi.org/10.1093/comjnl/6.4.308`.

[36] F. Lang. Explaining the lazy Krivine machine using explicit substitution and addresses. *Higher-Order and Symbolic Computation*, 20(3):257–270, 2007.

[37] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990. URL `http://gallium.inria.fr/ xleroy/publi/ZINC.pdf`.

[38] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998.

[39] J. Maraist, M. Odersky, D. N. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theor. Comput. Sci.*, 228(1-2):175–210, 1999.

[40] G. Mascari and M. Pedicini. Head linear reduction and pure proof net extraction. *Theor. Comput. Sci.*, 135(1):111–137, 1994.

[41] R. Milner. Local bigraphs and confluence: Two conjectures. *Electr. Notes Theor. Comput. Sci.*, 175(3):65–73, 2007.

[42] R. P. Nederpelt. The fine-structure of lambda calculus. Technical Report CSN 92/07, Eindhoven Univ. of Technology, 1992.

[43] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[44] P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Program*, 7(3): 231–264, 1997.