A Diamond Machine for Strong Evaluation

Beniamino Accattoli, Pablo Barenbaum

A Diamond Machine for Strong Evaluation

Beniamino Accattoli[1][0000−0003−4944−9944] and Pablo Barenbaum[2,3]

[1] Inria & LIX, École Polytechnique, UMR 7161, Palaiseau, France
[2] Universidad Nacional de Quilmes (CONICET), Bernal, Argentina
[3] CONICET-Universidad de Buenos Aires, Instituto de Ciencias de la Computación, Argentina

Abstract. Abstract machines for strong evaluation of the λ-calculus enter into arguments and have a set of transitions for backtracking out of an evaluated argument. We study a new abstract machine which avoids backtracking by splitting the run of the machine in smaller *jobs*, one for argument, and that jumps directly to the next job once one is finished. Usually, machines are also deterministic and implement deterministic strategies. Here we weaken this aspect and consider a light form of non-determinism, namely the *diamond property*, for both the machine and the strategy. For the machine, this introduces a modular management of jobs, parametric in a scheduling policy. We then show how to obtain various strategies, among which leftmost-outermost evaluation, by instantiating in different ways the scheduling policy.

Keywords: Lambda calculus, abstract machines, strong evaluation.

1 Introduction

An abstract machine for the λ-calculus, or for one of its extensions, is an implementation schema for a fixed evaluation strategy $\to_{\mathtt{str}}$ with sufficiently atomic operations (accounting for the *machine* part) and without too many implementative details (accounting for the *abstract* part). An abstract machine for $\to_{\mathtt{str}}$, ideally, refines the reduction of $\to_{\mathtt{str}}$-redexes realizing the following three tasks:

1. *Search*: searching for $\to_{\mathtt{str}}$-redexes;
2. *Names*: avoiding variable captures through some mechanism implementing α-conversion, or allowing one to avoid α-conversion altogether;
3. *Substitution*: refining meta-level substitution with an approximation based on delaying the substitution, which boils down to adopting a form of sharing, and replacing one variable occurrence at a time, in a demand-driven fashion.

These tasks are usually left to *meta-level operations* in λ-calculi, meaning that they happen outside the syntax of the calculus itself, in a black-box manner. The role of abstract machines is to explicitly take care of these aspects, or at least of some of them, reifying them from the meta-level to the object-level. Concretely, this is obtained by enriching the specification of the operational semantics with dedicated data structures. Additionally, such a specification is usually designed as to be efficient, and usually the evaluation strategy $\to_{\mathtt{str}}$ is deterministic.

Search, Backtracking, and Jumping. A first motivation of this paper is obtaining a better understanding of the search mechanism of abstract machines. When pushed at the meta-level, search is usually specified via deduction rules or via a grammar of evaluation contexts, assuming that, at each application of a rewriting rule, the term is correctly split into an evaluation context and a redex. The meta-level aspect is the fact that the process of splitting the term (or applying the deductive rules) is not taken into account as an operation of the calculus.

For simple evaluation strategies such as, for instance, weak call-by-name in the pure λ-calculus (also known as *weak head reduction*), abstract machines (such as the Krivine or the Milner abstract machine) have one search transition for every production of the evaluation contexts for the meta-level definition of search. For less simple strategies, the searching for redexes by the machine often also accounts for the further mechanism of *backtracking search*, that is not visible in the operational semantics, not even at meta-level. Such a form of search—which is completely unrelated to *backtracking-as-in-classical-logic*—happens when the machine has finished evaluating a sub-term and needs to backtrack to retrieve the next sub-term to inspect. Typically, this happens when implementing strategies that evaluate arguments, and in particular for strategies evaluating to *strong* normal form (that is, also under abstraction), the paradigmatic and simplest example of which is leftmost(-outermost[4]) evaluation.

As an example, let f be a strong normal form and consider executing a machine for leftmost evaluation on $\lambda x.xft$. The machine would go under $\lambda x.$, find the head variable x and then start searching for a β-redex in f. Since there are none, the machine arrives at the end of f and then it usually *backtracks* through the structure of f, as to exit it and then start searching inside t. Backtracking search is natural when one sees the searching process as a *walk* over the code, moving only between *adjacent* constructors. This is the dominating approach in the design of abstract machines for λ-calculi, since the entirety of the (small) literature on machines for strong evaluation adopts it [20,24,4,23,14,12,8,15,16].

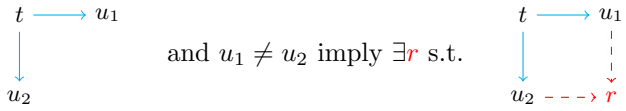
There is, however, a natural alternative approach which is *saving* the position where one would next backtrack, and then directly *jumping* to that position, instead of walking back to it. In this paper, we explore how to avoid backtracking search by adopting a form of *jumping search*. The idea is embarrassingly simple: creating a new *job* when an argument ready to be evaluated is found, adding it to a pool of jobs; then when the active job terminates, jumping to another job in the pool, without backtracking out of the terminated one.

Diamond Non-Determinism. A second motivation of the paper is to understand how to deal with diamond non-determinism at the level of machines. It is often the case that a deterministic strong strategy can be relaxed as to be non-deterministic. For instance, on the head normal form $x\,t\,u\,r$ leftmost evaluation would first evaluate t, then u, and then r. But the evaluations of t, u, and r are in fact independent, so that one could evaluate them in any order. One could even interleave their evaluations, as it is done for instance by the least

[4] To ease the language, in the paper we shorten *leftmost-outermost* to *leftmost*.

level strategy, a non-deterministic strong strategy coming from the linear logic literature, introduced—we believe—by Girard [25] and studied for instance by de Carvalho et al. [19] on proof nets and by Accattoli et al. [9] on the λ-calculus.

Such a form of non-determinism is benign, as it does not affect the result, nor the length of the evaluation. Abstractly, it is captured by the *diamond property* (here defined following Dal Lago and Martini [21], while Terese [32] defines it more restrictively, without requiring $u_1 \neq u_2$), the strongest form of confluence:



What makes it stronger than confluence is that both the opening span from t and the closing one to r are made of *single* steps, not of sequences of steps.

The diamond property can be seen as a liberal form of determinism, because—when it holds—all reductions to normal form have the same length, and if one such reduction exists then there are no diverging reductions.

External Strategy and Machine. Here we introduce a relaxed, diamond version of the leftmost strategy, which we deem *external strategy*. We are inspired by Accattoli et al. [8], who study a similar strategy for strong call-by-value.

Diamond strategies can be seen as uniform frameworks capturing different deterministic strategies (for instance the leftmost and the least level strategy). Accordingly, a non-deterministic machine implementing a diamond strategy would factor out the commonalities of different deterministic machines.

We then design a machine for the external strategy, the *EXternal Abstract Machine* (EXAM) by building over the jumping search explained above. The idea, again, is very simple. It amounts to relaxing the scheduling of jobs from the pool, by allowing the machine to non-deterministically select whatever unfinished job at each step, instead of having to terminate the active job and then having to move to the next one in the pool.

In fact, we go one step further. We define a *pool interface* and the definition of the EXAM is *abstract* in that it only refers to the interface. Then one can define different *pool templates* that implement various scheduling policies for jobs. The external strategy is implemented when adopting the most general, non-deterministic template. By only replacing the template, we show how the same machine can also implement the leftmost strategy. At the end of the paper, we also quickly overview a template for the least level strategy, as well as one for a fair strategy.

Related Work. This work adopts terminologies and techniques from the work on abstract machines by Accattoli and coauthors [3,4,1,5,7,10,8], and in particular refines their machine for leftmost evaluation [4]. They focus on the complexity of machines, while here we focus on *search* and ignore complexity, since their work shows that search has *linear* cost (in the time cost model, i.e. the number of β-steps) and thus it does not affect the asymptotic behavior, which is instead

linked to how the machine realizes the orthogonal *substitution* task. The study of machines for strong evaluation is a blind spot of the field, despite the relevance for the implementation of proof assistants. The few studies in the literature have all been cited above. Search for abstract machines is related to Danvy and Nielsen's (generalized) *refocusing* [22,17], which however has never dealt with the jumping search introduced here. General non-deterministic machines are studied by Biernacka et al. [13], but the setting is different as they are not diamond.

Proofs. Proofs are in the technical report [2].

2 Normal Forms and the Importance of Being External

Basics of λ. The set $\mathcal{T}$ of untyped λ-terms is defined by $t ::= x \mid \lambda x.\, t \mid t\, t$. The capture-avoiding substitution of x by u in t is written $t\{x := u\}$. The relation of β-reduction at the root $\mapsto_\beta \subseteq \mathcal{T} \times \mathcal{T}$ is defined by $(\lambda x.\, t)\, u \mapsto_\beta t\{x := u\}$.

We shall use various notions of contexts, which are terms with a single occurrence of a free variable $\langle\cdot\rangle$, called a *hole*. If C is a context, $C\langle t\rangle$ denotes the *plugging* of t in C which is the textual substitution of $\langle\cdot\rangle$ by t in C. Plugging might capture variables, for instance if $C = \lambda x.\lambda y.\langle\cdot\rangle$ then $C\langle xy\rangle = \lambda x.\lambda y.xy$. Note that instead one would have $(\lambda x.\lambda y.z)\{z := xy\} = \lambda x'.\lambda y'.xy$.

The relation of β-reduction $\to_\beta \subseteq \mathcal{T} \times \mathcal{T}$ is the context closure of $\mapsto_\beta$, i.e. $C\langle t\rangle \to_\beta C\langle u\rangle$ if $t \mapsto_\beta u$, compactly noted also as $\to_\beta := C\langle \mapsto_\beta\rangle$. An evaluation $e : t \to_\beta^* u$ is a possibly empty sequence of β-steps.

Proposition 1 (Normal forms). *β-Normal forms are described by:*

$$\text{Neutral terms} \quad n ::= x \mid nf \qquad\qquad \text{Normal forms} \quad f ::= n \mid \lambda x.f$$

Weak Head Reduction and External Redexes. The simplest evaluation strategy is weak head reduction $\to_{wh}$, which is obtained as the closure $A\langle \mapsto_\beta\rangle$ of the root β-rule $\mapsto_\beta$ by the following notion of applicative contexts:

$$\text{Applicative contexts} \quad A ::= \langle\cdot\rangle \mid A\, t.$$

Example: $(\lambda x.t)ur \to_{wh} t\{x{\leftarrow}u\}r$. Weak head reduction is deterministic. It fails to compute β-normal forms because it does not reduce arguments nor abstraction bodies, indeed $r((\lambda x.t)u) \not\to_{wh} r(t\{x{\leftarrow}u\})$ and $\lambda y.((\lambda x.t)u) \not\to_{wh} \lambda y.t\{x{\leftarrow}u\}$.

The key property of the weak head redex is that it is *external*, a key concept from the advanced rewriting theory of the λ-calculus studied by many authors [27,28,29,26,30,11,18,31,32,6]. Following Accattoli et al. [6], the intuition is that a redex R of a term t is *external* if:

1. *Action constraint*: no other redex in t can act on (that is, can erase or duplicate) R, and
2. *Hereditary clause*: the same it is true, hereditarily, for all the residuals of R after any other redex.

In $\delta(\mathtt{I}x)$, where $\delta := \lambda y.yy$ is the duplicator combinator and $\mathtt{I} := \lambda z.z$ is the identity combinator, the redex $\mathtt{I}x$ can be duplicated by δ, so it is not external because the action constraint is not respected. In $\mathtt{I}\delta(\mathtt{I}x)$, instead, the redex $\mathtt{I}x$ respects the action constraint, because $\mathtt{I}x$ is an outermost redex, and yet $\mathtt{I}x$ is not external because it does not validate the hereditary clause: its only residual after the step $\mathtt{I}\delta(\mathtt{I}x) \to_\beta \delta(\mathtt{I}x)$ can be duplicated by δ.

Defining external redexes requires the introduction of the theory of residuals, which is heavy and beyond the scope of this paper. The intuition behind it however guides the study in this paper, and we consider it a plus—rather than a weakness—that this can be done circumventing the theory of residuals.

Leftmost Reduction. One way to extend weak head reduction to compute β-normal forms as to always reduce external redexes is provided by *leftmost-outermost reduction* $\to_{lo}$ (shortened to *leftmost*). The definition relies on the notion of neutral term n used to describe normal forms, and it is given by the closure $L\langle\mapsto_\beta\rangle$ of root β by the following notion of leftmost contexts, defined by mutual induction with neutral contexts:

$$\textsc{Neutral ctxs}\quad N ::= \langle\cdot\rangle \mid nL \mid Nt \qquad \textsc{Leftmost ctxs}\ L ::= N \mid \lambda x.L$$

Some examples: $y((\lambda x.t)u) \to_{lo} y(t\{x{\leftarrow}u\})$ and $\lambda y.((\lambda x.t)u) \to_{lo} \lambda y.t\{x{\leftarrow}u\}$ but $\mathtt{I}y((\lambda x.t)u) \not\to_{lo} \mathtt{I}y(t\{x{\leftarrow}u\})$. Leftmost reduction is deterministic and *normalizing*, that is, if t has a reduction to normal form f then leftmost reduction reaches f. Formally, if $t \to_\beta{}^* f$ with f normal then $t \to_{lo}^* f$—for a recent simple proof of this classic result see Accattoli et al. [9]. The normalization property can be seen as a consequence of the fact that the strategy reduces only external redexes. Note that the outermost strategy (that reduces redexes not contained in any other redex) is instead not normalizing, as the following Ω redex is outermost (but not leftmost), where $\Omega := \delta\delta$ is the paradigmatic looping λ-term:

$$(\lambda x.\, \lambda y.\, x)\, z\, \Omega \to_\beta (\lambda x.\, \lambda y.\, x)\, z\, \Omega \to_\beta \dots \tag{1}$$

The key point is that the outermost strategy does reduce redexes that cannot be acted upon, but it does not satisfy the hereditary clause in the intuitive definition of external redex given above, for which one also needs an additional requirement such as selecting the leftmost redex among the outermost ones.

External Reduction. It is possible to define a strategy relaxing leftmost reduction, still reducing only external redexes, what we call *external reduction*. The definition uses the auxiliary notions of rigid terms and contexts, plus the applicative contexts A used for weak head reduction. The terminology is inspired by Accattoli et al.'s similar strategy for strong call-by-value evaluation [8].

Definition 1. *The following categories of terms and contexts are defined mutually inductively by the grammar:*

$$\begin{array}{lll} \textsc{Rigid terms} & r ::= x \mid r\,t & \\ \textsc{Rigid ctxs} & R ::= \langle\cdot\rangle \mid r\,E \mid R\,t & \textsc{External ctxs}\quad E ::= R \mid \lambda x.\,E \end{array}$$

External reduction *is the rewriting relation* $\to_{\mathtt{x}} \subseteq \mathcal{T} \times \mathcal{T}$ *on* λ*-terms defined as the closure of root* β*-reduction under external contexts, that is,* $\to_{\mathtt{x}} := E\langle \mapsto_\beta \rangle$.

Alternative streamlined definitions for these notions are:

$$\begin{array}{ll} r & ::= x\, t_1 \dots t_n \\ R & ::= \langle\cdot\rangle\, t_1 \dots t_n \mid x\, u_1 \dots u_m\, E\, t_1 \dots t_n \qquad E ::= \lambda x_1 \dots x_k.\, R \end{array}$$

As proved below, the leftmost strategy is a special case of the external one. The converse does *not* hold: $t = x(\mathtt{I}y)(\mathtt{I}z) \to_{\mathtt{x}} x(\mathtt{I}y)z = u$ but $t \not\to_{lo} u$. Instead, $t \to_{lo} xy(\mathtt{I}z) = r$. Note also a case of diamond: $t \to_{\mathtt{x}} r$ and $r \to_{\mathtt{x}} xyz\ {}_{\mathtt{x}}\!\leftarrow u$.

Proposition 2 (Properties of external reduction).

1. Leftmost is external*: if* $t \to_{lo} u$ *then* $t \to_{\mathtt{x}} u$.
2. External diamond*: if* $u\ {}_{\mathtt{x}}\!\leftarrow \cdot \to_{\mathtt{x}} r$ *with* $u \neq r$ *then* $u \to_{\mathtt{x}} \cdot\ {}_{\mathtt{x}}\!\leftarrow r$.

3 Preliminaries: Abstract Machines

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

- *States.* A state $s \in \mathtt{States}$ is composed by the *active term* t, and some data structures. Terms in states are actually pre-terms.
- *Transitions.* The pair $(\mathtt{States}, \rightsquigarrow)$ is a transition system with transitions $\rightsquigarrow$ partitioned into β*-transitions* $\rightsquigarrow_\beta$ (usually just one), that are meant to correspond to β-steps, and *overhead transitions* $\rightsquigarrow_{\mathtt{o}}$, that take care of the various tasks of the machine (searching, substituting, and α-renaming).
- *Initialization.* The component $\triangleleft \subseteq \Lambda \times \mathtt{States}$ is the *initialization relation* associating λ-terms to initial states. It is a *relation* and not a function because $t \triangleleft s$ maps a λ-term t (considered modulo α) to a state s having a *pre-term representant* of t (which is not modulo α) as active term. Intuitively, any two states s and s' such that $t \triangleleft s$ and $t \triangleleft s'$ are α-equivalent. A state s is *reachable* if it can be reached starting from an initial state, that is, if $s' \rightsquigarrow^* s$ where $t \triangleleft s'$ for some t and s', which we abbreviate using $t \triangleleft s' \rightsquigarrow^* s$.
- *Read-back.* The read-back function $\underline{\cdot} : \mathtt{States} \to \Lambda$ turns reachable states into λ-terms and satisfies the *initialization constraint*: if $t \triangleleft s$ then $\underline{s} =_\alpha t$.

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

Well-Namedness and Renaming. For the machines at work in this paper, the pre-terms in initial states shall be *well-named*, that is, they have pairwise distinct bound names; for instance $(\lambda x.x)(\lambda y.yy)$ is well-named while $(\lambda x.x)(\lambda x.xx)$ is not. We shall also write $t^{\mathtt{R}}$ in a state s for a *fresh well-named renaming* of t, *i.e.* $t^{\mathtt{R}}$ is α-equivalent to t, well-named, and its bound variables are fresh with respect to those in t and in the other components of s.

Implementation Theorem, Abstractly. We now formally define the notion of a machine implementing a strategy.

Definition 2 (Machine implementation). *A machine* $\mathtt{M} = (\mathtt{States}, \rightsquigarrow, \cdot\triangleleft\cdot, \underline{\cdot})$ implements a strategy $\rightarrow_{\mathtt{str}}$ *when given a* λ*-term* t *the following holds:*

1. Runs to evaluations*: for any* $\mathtt{M}$*-run* $\rho : t \triangleleft s \rightsquigarrow^*_{\mathtt{M}} s'$ *there exists a* $\rightarrow_{\mathtt{str}}$*-evaluation* $e : t \rightarrow^*_{\mathtt{str}} \underline{s'}$*.*
2. Evaluations to runs*: for every* $\rightarrow_{\mathtt{str}}$*-evaluation* $e : t \rightarrow^*_{\mathtt{str}} u$ *there exists a* $\mathtt{M}$*-run* $\rho : t \triangleleft s \rightsquigarrow^*_{\mathtt{M}} s'$ *such that* $\underline{s'} = u$*.*
3. β-Matching*: in both previous points the number* $|\rho|_\beta$ *of* β*-transitions in* ρ *is exactly the length* $|e|$ *of the evaluation* e*,* i.e. $|e| = |\rho|_\beta$*.*

Next, we give sufficient conditions that a machine and a strategy have to satisfy in order for the former to implement the latter, what we call *an implementation system*. In the literature, strategies and machines are usually assumed to be deterministic. In Accattoli et al. [8], there is the case of a deterministic machine implementing a diamond strategy. Here we shall have a diamond machine implementing a diamond strategy, which is why the requirements are a bit different than for previous notion of implementation systems in the literature [10,7,8].

Definition 3 (Implementation system). *A machine* $\mathtt{M} = (\mathtt{States}, \rightsquigarrow, \cdot\triangleleft\cdot, \underline{\cdot})$ *and a strategy* $\rightarrow_{\mathtt{str}}$ *form an* implementation system *if:*

1. Overhead transparency*:* $s \rightsquigarrow_{\mathtt{o}} s'$ *implies* $\underline{s} = \underline{s'}$*;*
2. β-projection*:* $s \rightsquigarrow_\beta s'$ *implies* $\underline{s} \rightarrow_{\mathtt{str}} \underline{s'}$*;*
3. Overhead termination*:* $\rightsquigarrow_{\mathtt{o}}$ *terminates;*
4. β-reflection*: if* s *is* $\rightsquigarrow_{\mathtt{o}}$*-normal and* $\underline{s} \rightarrow_{\mathtt{str}} u$ *then there exists* s' *such that* $s \rightsquigarrow_\beta s'$ *and* $\underline{s'} = u$*.*

The first two properties guarantee that the *runs to evaluations* part of the implementation holds, the third and fourth properties instead induce the *evaluation to runs* part, which is slightly more delicate. In the deterministic case, such a second part usually follows from a weaker notion of implementation system, where β-reflection is replaced by the weaker *halt property*, stating that *if* s *is final then* $\underline{s}$ *is normal*. The diamond aspect of our study requires the stronger β-reflection property, which actually subsumes the halt one. Indeed, if $\underline{s}$ is not normal then by β-reflection s is not final.

Thanks to a simple lemma for the *evaluation to runs* part (in the technical report [2]), we obtain the following abstract implementation theorem.

Theorem 1 (Sufficient condition for implementations). *Let* $\mathtt{M}$ *be a machine and* $\rightarrow_{\mathtt{str}}$ *be a strategy forming an implementation system. Then,* $\mathtt{M}$ *implements* $\rightarrow_{\mathtt{str}}$*. More precisely,* β*-projection and overhead transparency imply the* runs to evaluations *part (plus* β*-matching), and overhead termination and* β*-reflection imply the* evaluations to runs *part (plus* β*-matching).*

Data structures, States, and Initialization

Stacks	$S, S' ::= \epsilon \mid t : S$	Environments	$E, E' ::= \epsilon \mid [x{\leftarrow}t] : E$
States	$s, s' ::= (t \mid S \mid E)$	Initialization	$t \triangleleft s$ if $s = (t^{\mathtt{R}} \mid \epsilon \mid \epsilon)$

Transitions

Active Term	Stack	Env		Active Term	Stack	Env
tu	S	E	$\rightsquigarrow_{\mathtt{sea}_@}$	t	$u : S$	E
$\lambda x.t$	$u : S$	E	$\rightsquigarrow_{\beta}$	t	S	$[x{\leftarrow}u] : E$
x	S	E	$\rightsquigarrow_{\mathtt{sub}}$	$E(x)^{\mathtt{R}}$	S	E
						If $x \in \mathtt{dom}\, E$

Fig. 1: Definition of the Milner Abstract Machine (MAM).

4 Preliminaries: The Milner Abstract Machine

The new machine for the external strategy that we are about to introduce builds on the Milner Abstract Machine (shortened to MAM) for weak head reduction by Accattoli et al. [3], that is probably the simplest abstract machine for the λ-calculus in the literature. In this section, we overview the MAM, the data structures and transitions of which are defined in Fig. 1.

Data Structures. The MAM has two data structures, the stack S and the environment E, which are lists. We use ':' for consing a single element onto a list, but also for list concatenation, so for instance $S : S'$ stands for the concatenation of stacks. The set of variables bound by an environment $E = [x_1{\leftarrow}t_1] \dots [x_k{\leftarrow}t_k]$ is $\{x_1, \dots, x_k\}$ and it is noted $\mathtt{dom}\, E$.

Transitions of the MAM. A term t is initialized into an initial state $t \triangleleft s$ by simply using an arbitrary well-named renaming $t^{\mathtt{R}}$ as active term together with empty stack and environment. The MAM *searches* for β-redexes in the active term by descending on the left of applications via transition $\rightsquigarrow_{\mathtt{sea}_@}$, while accumulating arguments on the (applicative) *stack*, which is simply a stack of terms. If it finds an abstraction $\lambda x.t$ and the stack has u on top, then it performs the machine analogous of a β-redex, that is a $\rightsquigarrow_\beta$ transition, which adds the entry $[x{\leftarrow}u]$ on top of the *environment*, to be understood as a delayed, explicit substitution. If the MAM finds a variable x, then it looks up in the environment E if it finds an associated entry $[x{\leftarrow}t]$, and replaces x with an α-renamed $t^{\mathtt{R}}$ copy of t.

Transitions $\rightsquigarrow_{\mathtt{sea}_@}$ and $\rightsquigarrow_{\mathtt{sub}}$ are the overhead transitions of the MAM, that is, $\rightsquigarrow_{\mathtt{o}} := \rightsquigarrow_{\mathtt{sea}_@, \mathtt{sub}}$, and $\rightsquigarrow_\beta$ is its only β-transition. The MAM is deterministic.

Read-Back. The read-back of MAM states to terms can be defined in at least two ways, by either first reading back the environment or the stack. Here we give an environment-first definition, which shall be used also for the EXAM.

Definition 4 (MAM read-back). *The read-back $\underline{t}_E$ and $\underline{S}_E$ of terms and stack with respect to an environment E are the terms and stacks given by:*

$$\begin{array}{rlll} \textsc{Terms} & \underline{t}_{\epsilon} := t & \underline{t}_{[x\leftarrow u]:E} := \underline{(t\{x := u\})}_{E} \\ \textsc{Stacks} & \underline{\epsilon}_{E} := \epsilon & \underline{t : S}_{E} := \underline{t}_{E} : \underline{S}_{E} \end{array}$$

The read-back $\underline{t}_S$ *of* t *with respect to a stack* S *is the term given by:*

$$\underline{t}_{\epsilon} := t \qquad \underline{t}_{u:S} := \underline{(t\,u)}_{S}$$

Finally, the read-back of a state is defined as $\underline{(t \mid S \mid E)} := \underline{\underline{t}_{E}}_{\underline{S}_{E}}$.

Theorem 2 ([3]). *The MAM implements weak head reduction* $\to_{wh}$.

Environments are defined as *lists* of entries, but they are meant to be concretely implemented as a store, without a rigid list structure. The idea is that variables are implemented as memory locations, as to obtain constant-time access to the right entry of the environment via the operation $E(x)$. It is nonetheless standard to define environments as lists, as it helps one stating invariants concerning them. For more implementative details, see Accattoli and Barras [5].

Comparison with the KAM. For the reader acquainted with the famous Krivine Abstract Machine (KAM), the difference is that the stack and the environment of the MAM contain *codes*, not *closures* as in the KAM, and that there is a single *global* environment instead of many *local* environments. A global environment indeed circumvents the complex mutually recursive notions of *local environment* and *closure*, at the price of the explicit α-renaming $t^{\mathtt{R}}$ which is applied *on the fly* in $\rightsquigarrow_{\mathtt{sub}}$. The price however is negligible, at least theoretically, as the asymptotic complexity of the machine is not affected, see Accattoli and co-authors [3,5] (the same can be said of variable names vs de Bruijn indexes/levels).

5 The External Abstract Machine

In this section, we define the EXternal Abstract Machine (EXAM), an abstract machine for the external strategy $\to_{\mathtt{x}}$, by using the MAM as a sort of building block. The EXAM is given in Fig. 2 and explained in the following paragraphs. An example of run is given at the end of this section.

Data Structures. The EXAM has three data structures, two of which are new with respect to the MAM:

- The *approximant* (of the normal form) $\mathbb{A}$, which collects the parts of the normal form already computed by the run of the EXAM. The approximant is a *named multi-context*, defined below, that is, a context with zero, one, or more named holes $\langle\cdot\rangle_\alpha$, each one identified by a distinct name α, β, etc.
- The *pool* P, which is a data structure containing a set of named MAM *jobs*, providing operations for scheduling the execution of these jobs. Each named job j_α has shape $(t, S)_\alpha$, that is, it contains a term and a stack. The idea is that the job $j_\alpha = (t, S)_\alpha$ of name α is executing the term corresponding to (t, S) and that the result of such execution shall be plugged in the approximant $\mathbb{A}$, replacing the hole $\langle\cdot\rangle_\alpha$. Pools are discussed in detail below.
- The *environment* E, which is as for the MAM except that it is shared among all the jobs in the pool.

Data structures, States, and Initialization

Approx.	$\mathbb{A} ::= \langle\cdot\rangle_\alpha \mid \mathbb{R} \mid \lambda x.\,\mathbb{A}$	Rigid approx.	$\mathbb{R} ::= x \mid \mathbb{R}\,\mathbb{A}$
Jobs	$j_\alpha ::= (t, S)_\alpha$	States	$s ::= [\![\mathbb{A} \mid P \mid E]\!]$ with P a pool
		Initialization	$t \triangleleft s$ if $s = [\![\langle\cdot\rangle_\alpha \mid \mathtt{new}((t^{\mathtt{R}}, \epsilon)_\alpha) \mid \epsilon]\!]$

Transitions

Ap.	Pool	Env		Ap.	Pool	Env
$\mathbb{A}$	$(t\,u, S)_\alpha \overset{\mathtt{sel}}{\leftarrow} P$	E	$\rightsquigarrow_{\mathtt{sea}_@}$	$\mathbb{A}$	$(t, u : S)_\alpha \overset{\mathtt{dro}}{\rightarrow} P$	E
$\mathbb{A}$	$(\lambda x.\,t, u : S)_\alpha \overset{\mathtt{sel}}{\leftarrow} P$	E	$\rightsquigarrow_\beta$	$\mathbb{A}$	$(t, S)_\alpha \overset{\mathtt{dro}}{\rightarrow} P$	$[x \leftarrow u] : E$
$\mathbb{A}$	$(x, S)_\alpha \overset{\mathtt{sel}}{\leftarrow} P$	E	$\rightsquigarrow_{\mathtt{sub}}$	$\mathbb{A}$	$(E(x)^{\mathtt{R}}, S)_\alpha \overset{\mathtt{dro}}{\rightarrow} P$	E
				If $x \in \mathtt{dom}\,E$ and $t^{\mathtt{R}}$ is a fresh renaming of t		
$\mathbb{A}$	$(\lambda x.\,t, \epsilon)_\alpha \overset{\mathtt{sel}}{\leftarrow} P$	E	$\rightsquigarrow_{\mathtt{sea}_\lambda}$	$\mathbb{A}'$	$(t, \epsilon)_\alpha \overset{\mathtt{dro}}{\rightarrow} P$	E
				With $\mathbb{A}' := \mathbb{A}\langle\lambda x.\,\langle\cdot\rangle_\alpha\rangle_\alpha$		
$\mathbb{A}$	$(x, t_1 : .. : t_n)_\alpha \overset{\mathtt{sel}}{\leftarrow} P$	E	$\rightsquigarrow_{\mathtt{sea}_\mathcal{V}}$	$\mathbb{A}'$	$(t_1, \epsilon)_{\beta_1} : .. : (t_n, \epsilon)_{\beta_n} \overset{\mathtt{add}}{\rightarrow} P$	E
				If $x \notin \mathtt{dom}\,E$, and with $n \geq 0$, $\mathbb{A}' := \mathbb{A}\langle x\,\langle\cdot\rangle_{\beta_1} .. \langle\cdot\rangle_{\beta_n}\rangle_\alpha$, and $\beta_1, .., \beta_n$ fresh		

Fig. 2: Definition of the EXternal Abstract Machine (EXAM).

Transitions and Functioning of the EXAM. A term t is initialized into an initial state $t \triangleleft s$ by creating a pool with a single named job $(t^{\mathtt{R}}, \epsilon)_\alpha$ (having a well-named $t^{\mathtt{R}}$ version of t and an empty stack) and pairing it with the approximant $\mathbb{A} = \langle\cdot\rangle_\alpha$ and empty environment. The EXAM proceeds as the MAM until it reaches a MAM final state. Let us consider the normal forms for weak head reduction and the corresponding final states of the MAM, which are of two kinds:

1. *Abstractions (with no arguments)*: the $\rightarrow_{wh}$ normal form is $\lambda x.u$ which is the read-back of a final MAM state $(\lambda x.t, \epsilon, E)$ with empty stack (that is, $u = \underline{t}_E$). In this case, the EXAM performs a $\rightsquigarrow_{\mathtt{sea}_\lambda}$ transition, storing $\lambda x.\langle\cdot\rangle_\alpha$ into the approximant $\mathbb{A}$ at α, and adding a named job $(t, \epsilon)_\alpha$ to the pool P.
2. *Possibly applied variables (with no substitution)*: the $\rightarrow_{wh}$ normal form is $x u_1 \dots u_n$ with $n \geq 0$, which is the read-back of a final state $(x, t_1 : \dots : t_n, E)$ with $x \notin \mathtt{dom}\,E$ (that is, $u_i = \underline{t_i}_E$). In this case, the EXAM performs a $\rightsquigarrow_{\mathtt{sea}_\mathcal{V}}$ transition. If $n = 0$ then $\rightsquigarrow_{\mathtt{sea}_\mathcal{V}}$ simply adds x into the approximant $\mathbb{A}$ at α. If $n > 0$ then $\rightsquigarrow_{\mathtt{sea}_\mathcal{V}}$ adds n new named jobs $(t_1, \epsilon)_{\beta_1}, .., (t_n, \epsilon)_{\beta_n}$ to the pool P and adds $x\langle\cdot\rangle_{\alpha_1} .. \langle\cdot\rangle_{\alpha_1}$ into the approximant $\mathbb{A}$ at α.

Transitions $\rightsquigarrow_{\mathtt{sea}_@}$, $\rightsquigarrow_{\mathtt{sea}_\lambda}$, and $\rightsquigarrow_{\mathtt{sea}_\mathcal{V}}$ are the search transitions of the EXAM. Together with $\rightsquigarrow_{\mathtt{sub}}$, they are the overhead transitions of the EXAM, that is, $\rightsquigarrow_{\mathtt{o}} := \rightsquigarrow_{\mathtt{sea}_@, \mathtt{sub}, \mathtt{sea}_\lambda, \mathtt{sea}_\mathcal{V}}$, and $\rightsquigarrow_\beta$ is its only β-transition. The transition relation of the EXAM is the union of all these relations, *i.e.* $\rightsquigarrow_{\mathtt{EXAM}} := \rightsquigarrow_{\mathtt{sea}_@, \beta, \mathtt{sub}, \mathtt{sea}_\lambda, \mathtt{sea}_\mathcal{V}}$.

Pool Interface and Templates. The EXAM selects at each step a (named) job from the pool—the one performing the step—according to a possibly non-deterministic policy, and drops it back in the pool after the transition, unless the job is over, which happens in transition $\rightsquigarrow_{\mathtt{sea}_\mathcal{V}}$. In general, *dropping a job back into*

a pool and *adding a job to a pool* are not the same operation, since the former applies to jobs that were in the pool before being selected, while addition is reserved to new jobs. We actually abstract away from a job scheduling policy and from the details of the pool data structure: the pool is an abstract *interface* which can be realized by various concrete data structures called *pool templates*.

Definition 5 (Pool templates). *A pool template is a data structure P coming with the following five operations of the* pool interface*:*

- Names, support, and new*: there are a name function* $\mathtt{names}(P) = \{\alpha_1, .., \alpha_n\}$ *providing the finite and possibly empty set of the names of the jobs in the pool (*$\mathbb{N} \ni n \geq 0$*), a support function* $\mathtt{supp}(P) = \{j_{\alpha_1}, .., j_{\alpha_n}\}$ *providing the set of jobs in the pool (indexed by* $\mathtt{names}(P)$*), and a function* $\mathtt{new}(j_\alpha)$ *creating a pool containing* j_α*, that is, such that* $\mathtt{supp}(\mathtt{new}(j_\alpha)) = \{j_\alpha\}$*.*
- Selection*: there is a selection relation* $\overset{\mathtt{sel}}{\leftharpoonup} (P, j_\alpha, P')$ *such that* $j_\alpha \in \mathtt{supp}(P)$ *and* $\mathtt{supp}(P') = \mathtt{supp}(P) \setminus \{j_\alpha\}$*. The intuition is that* P' *is* P *without* j_α*, which has been selected and removed from* P*. There is a* choice constraint*: if* P *is non-empty then* $\overset{\mathtt{sel}}{\leftharpoonup} (P, j_\alpha, P')$ *holds for some* j_α *and* P'*. We write* $j_\alpha \overset{\mathtt{sel}}{\leftharpoonup} P'$ *for a pool* P *such that* $\overset{\mathtt{sel}}{\leftharpoonup} (P, j_\alpha, P')$*.*
- Dropping*: there is a dropping function* $\overset{\mathtt{dro}}{\rightharpoonup} (j_\alpha, P) = P'$ *defined when* $\alpha \notin \mathtt{names}(P)$ *and such that* $\mathtt{supp}(P') = \mathtt{supp}(P) \cup \{j_\alpha\}$*. Dropping is meant to add a job* j_α *back to a pool* P *from which* j_α *was previously selected. It is not necessarily the inverse of selection. We write* $j_\alpha \overset{\mathtt{dro}}{\rightharpoonup} P$ *for the pool* $\overset{\mathtt{dro}}{\rightharpoonup} (j_\alpha, P)$*.*
- Adding*: similarly, there is an adding function* $\overset{\mathtt{add}}{\rightharpoonup} (j_\alpha, P) = P'$ *defined when* $\alpha \notin \mathtt{names}(P)$ *and such that* $\mathtt{supp}(P') = \mathtt{supp}(P) \cup \{j_\alpha\}$*. Adding is meant to add a new job* j_α *to a pool* P*, that is, a job that has never been in* P*. We write* $j_\alpha \overset{\mathtt{add}}{\rightharpoonup} P$ *for* $\overset{\mathtt{add}}{\rightharpoonup} (j_\alpha, P) = P'$*, and extend it to lists as follows:* $\epsilon \overset{\mathtt{add}}{\rightharpoonup} P := P$*, and* $j_{\alpha_1} : .. : j_{\alpha_n} \overset{\mathtt{add}}{\rightharpoonup} P := j_{\alpha_n} \overset{\mathtt{add}}{\rightharpoonup} \left(j_{\alpha_1} : .. : j_{\alpha_{n-1}} \overset{\mathtt{add}}{\rightharpoonup} P\right)$*.*

Set EXAM. The simplest pool template is the *set template* where pools P are sets of named jobs, the support is the pool itself (and the name set is as expected), $\mathtt{new}(j_\alpha)$ creates a singleton with j_α, selection is the relation $\{(P, j_\alpha, P \setminus \{j_\alpha\}) \,|\, j_\alpha \in P\}$, and both dropping and adding are the addition of an element. The set template models the most general behavior, as *any* job of the pool can then be selected for the next step. The EXAM instantiated on the set template is called *Set EXAM*. Other templates shall be considered at the end of the paper, motivating in particular the distinction between dropping and adding.

Approximants and Named Multi-Contexts. The definition of the EXAM rests on approximants, which are stable prefixes of normal forms, that is, normal forms from which some sub-terms have been removed and replaced with named holes. In fact, we are going to introduce more general *(named) multi-contexts* to give a status to approximants in which some but not all holes have been replaced by an *arbitrary term*—which shall be needed in proofs (when manipulating the read-back)—thus losing their "normal prefix" property.

Definition 6 (Named multi-contexts). *A* (named) multi-context $\mathbb{C}$ *is a* λ*-term in which there may appear free occurrences of (named) holes,* i.e.*:*

$$\text{(Named) Multi-contexts} \quad \mathbb{C} ::= x \mid \langle\cdot\rangle_\alpha \mid \lambda x.\,\mathbb{C} \mid \mathbb{C}\,\mathbb{C}$$

The plugging $\mathbb{C}\langle\mathbb{C}'\rangle_\alpha$ *of* α *by* $\mathbb{C}'$ *in* $\mathbb{C}$*, is the capture-allowing substitution of* $\langle\cdot\rangle_\alpha$ *by* $\mathbb{C}'$ *in* $\mathbb{C}$*. We write* $\mathtt{names}(\mathbb{C})$ *for the set of names that occur in* $\mathbb{C}$*. We shall use only multi-contexts where named holes have pairwise* distinct names*.*

Note that a multi-context $\mathbb{C}$ without holes is simply a term, thus the defined notion of plugging subsumes the plugging $\mathbb{C}\langle t\rangle_\alpha$ of terms in multi-contexts.

Approximants $\mathbb{A}$ are defined in Fig. 2 by mutual induction with rigid approximants $\mathbb{R}$, and are special cases of multi-contexts. Alternative streamlined definitions for (rigid) approximants are (possibly $y = x_i$ for some $i \in \{1,\dots,n\}$):

$$\mathbb{R} ::= x\,\mathbb{A}_1..\mathbb{A}_n \qquad \mathbb{A} ::= \lambda x_1..x_n.\,\langle\cdot\rangle_\alpha \mid \lambda x_1..x_n.\,y\,\mathbb{A}_1..\mathbb{A}_n$$

Note that in $\mathbb{A}$ and $\mathbb{R}$ holes are never applied, that is, they are *non-applying* multi-contexts. For the sake of readability, in the paper we only give statements about approximants, which are then reformulated in the technical report [2] by pairing them with a similar statement about rigid approximants, and proving the two of them simultaneously by mutual induction.

We prove two properties of approximants. Firstly, to justify that transitions $\rightsquigarrow_{\mathtt{sea}_\lambda}$ and $\rightsquigarrow_{\mathtt{sea}_\mathcal{V}}$ are well-defined, we show that the first component of the state on their right-hand side is indeed an approximant. Secondly, we relate approximants with normal forms, to justify the terminology.

Lemma 1 (Inner extension of approximants). *If* $\mathbb{A}$ *is an approximant and* $\beta_1,\dots,\beta_n \notin \mathtt{names}(\mathbb{A})$ *then* $\mathbb{A}\langle\lambda x.\,\langle\cdot\rangle_\alpha\rangle_\alpha$ *and* $\mathbb{A}\langle x\,\langle\cdot\rangle_{\beta_1}..\langle\cdot\rangle_{\beta_n}\rangle_\alpha$ *are approximants.*

Lemma 2. *An approximant* $\mathbb{A}$ *without named holes is a normal form.*

Read-Back. To give a notion of read-back that is independent of the pool template, we define the read-back using a set X of uniquely named jobs—standing for the support $\mathtt{supp}(P)$ of the pool—rather than the pool P itself. Moreover, we need a way of applying the substitution induced by an environment to named jobs and sets of named jobs, which is based on the notions $\underline{t}_E$ and $\underline{S}_E$ for terms and stacks given for the MAM, from which we also borrow the definition of $\underline{t}_S$.

Definition 7 (EXAM read-back). *Applying an environment* E *to jobs and job sets is defined as follows:*

$$\text{Jobs/jobs sets} \quad \underline{(t,S)_\alpha}_E := (\underline{t}_E, \underline{S}_E)_\alpha \qquad \underline{\{j_{\alpha_1},..,j_{\alpha_n}\}}_E := \{\underline{j_{\alpha_1}}_E,..,\underline{j_{\alpha_n}}_E\}$$

The read-back of jobs, and of a multi context $\mathbb{C}$ *with respect to a set of uniquely named jobs* $\{j_{\alpha_1},..,j_{\alpha_n}\}$ *are defined as follows:*

$$\underline{(t,S)_\alpha} := \underline{t}_S \qquad \underline{\mathbb{C}}_{\{j_{\alpha_1},..,j_{\alpha_n}\}} := \mathbb{C}\langle\underline{j_{\alpha_1}}\rangle_{\alpha_1}..\langle\underline{j_{\alpha_n}}\rangle_{\alpha_n}$$

An EXAM state s *is read-back as a multi-context setting* $\underline{[\![\mathbb{A} \mid P \mid E]\!]} := \underline{\mathbb{A}}_{\underline{\mathtt{supp}(P)}_E}$.

Diamond. Since the selection operation is non-deterministic, the EXAM in general is non-deterministic. The most general case is given by the *Set EXAM*, which is the EXAM instantiated with the set template for pools described after Definition 5. As for the external strategy, the Set EXAM has the diamond property up to a slight glitch: swapping the order of two β-transitions on two different jobs, adds entries to the environment in different orders.

Let $\approx$ be the minimal equivalence relation on environments containing the following relation:

$$E : [x{\leftarrow}t] : [y{\leftarrow}u] : E' \quad \sim \quad E : [y{\leftarrow}u] : [x{\leftarrow}t] : E' \quad \text{if } x \notin u \text{ and } y \notin t$$

Let $\equiv$ be the relation over states s. t. $[\![\mathbb{A} \mid P \mid E_1]\!] \equiv [\![\mathbb{A} \mid P \mid E_2]\!]$ if $E_1 \approx E_2$.

Proposition 3. *The Set EXAM is diamond up to $\equiv$, i.e., if $s \rightsquigarrow_{\texttt{EXAM}} s_1$ and $s \rightsquigarrow_{\texttt{EXAM}} s_2$ then $\exists s_1'$ and s_2' such that $s_1 \rightsquigarrow_{\texttt{EXAM}} s_1'$, $s_2 \rightsquigarrow_{\texttt{EXAM}} s_2'$, and $s_1' \equiv s_2'$.*

Example 1. The following is a possible run of the Set EXAM—that is, the EXAM with the set template for pools—on the term $t := x(\mathtt{I}_y z)(\delta_w z)$ where $\mathtt{I}_y = \lambda y.y$ and $\delta_w = \lambda w.ww$, ending in a final state.

Approx.	Pool	Env	Tran.	Selected Job
$\langle\cdot\rangle_\alpha$	$\{(x(\mathtt{I}_y z)(\delta_w z), \epsilon)_\alpha\}$	ϵ	$\rightsquigarrow_{\texttt{sea}_@}$	α
$\langle\cdot\rangle_\alpha$	$\{(x(\mathtt{I}_y z), \delta_w z)_\alpha\}$	ϵ	$\rightsquigarrow_{\texttt{sea}_@}$	α
$\langle\cdot\rangle_\alpha$	$\{(x, \mathtt{I}_y z : \delta_w z)_\alpha\}$	ϵ	$\rightsquigarrow_{\texttt{sea}_\mathcal{V}}$	α
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$\{(\mathtt{I}_y z, \epsilon)_\beta, (\delta_w z, \epsilon)_\gamma\}$	ϵ	$\rightsquigarrow_{\texttt{sea}_@}$	γ
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$\{(\mathtt{I}_y z, \epsilon)_\beta, (\delta_w, z)_\gamma\}$	ϵ	$\rightsquigarrow_\beta$	γ
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$\{(\mathtt{I}_y z, \epsilon)_\beta, (ww, \epsilon)_\gamma\}$	$[w{\leftarrow}z]$	$\rightsquigarrow_{\texttt{sea}_@}$	β
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$\{(\mathtt{I}_y, z)_\beta, (ww, \epsilon)_\gamma\}$	$[w{\leftarrow}z]$	$\rightsquigarrow_{\texttt{sea}_@}$	γ
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$\{(\mathtt{I}_y, z)_\beta, (w, w)_\gamma\}$	$[w{\leftarrow}z]$	$\rightsquigarrow_\beta$	β
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$\{(y, \epsilon)_\beta, (w, w)_\gamma\}$	$[y{\leftarrow}z] : [w{\leftarrow}z]$	$\rightsquigarrow_{\texttt{sub}}$	β
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$\{(z, \epsilon)_\beta, (w, w)_\gamma\}$	$[y{\leftarrow}z] : [w{\leftarrow}z]$	$\rightsquigarrow_{\texttt{sub}}$	γ
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$\{(z, \epsilon)_\beta, (z, w)_\gamma\}$	$[y{\leftarrow}z] : [w{\leftarrow}z]$	$\rightsquigarrow_{\texttt{sea}_\mathcal{V}}$	γ
$x\langle\cdot\rangle_\beta(x\langle\cdot\rangle_{\gamma'})$	$\{(z, \epsilon)_\beta, (w, \epsilon)_{\gamma'}\}$	$[y{\leftarrow}z] : [w{\leftarrow}z]$	$\rightsquigarrow_{\texttt{sub}}$	γ'
$x\langle\cdot\rangle_\beta(x\langle\cdot\rangle_{\gamma'})$	$\{(z, \epsilon)_\beta, (z, \epsilon)_{\gamma'}\}$	$[y{\leftarrow}z] : [w{\leftarrow}z]$	$\rightsquigarrow_{\texttt{sea}_\mathcal{V}}$	β
$xz(x\langle\cdot\rangle_{\gamma'})$	$\{(z, \epsilon)_{\gamma'}\}$	$[y{\leftarrow}z] : [w{\leftarrow}z]$	$\rightsquigarrow_{\texttt{sea}_\mathcal{V}}$	γ'
$xz(zz)$	$\emptyset$	$[y{\leftarrow}z] : [w{\leftarrow}z]$		

6 Runs to Evaluations

In this section, we develop the projection of EXAM runs on external evaluations, and then instantiates it with a deterministic pool template obtaining runs corresponding to leftmost evaluations.

Overhead Transparency. By the abstract recipe for implementation theorems in Sect. 3, to project runs on evaluations we need to prove *overhead transparency* and *β-projection*. Overhead transparency is simple, it follows from the definition of read-back plus some of its basic properties (in the technical report [2]).

Proposition 4 (Overhead transparency). *If $s \rightsquigarrow_{\mathtt{o}} s'$ then $\underline{s} = \underline{s'}$.*

Invariants. To prove the β-projection property, we need some invariants of the EXAM. We have a first set of invariants concerning variable names, hole names, and binders. A notable point is that their proofs use only properties of the pool interface, and are thus valid for every pool template instantiation of the EXAM.

Terminology: a *binding occurrence* of a variable x is an occurrence of $\lambda x.\, t$ in $\mathbb{A}$, P or E, or an occurrence of $[x \leftarrow t]$ in E, for some t.

Lemma 3 (EXAM Invariants). *Let $s = [\![\mathbb{A} \mid P \mid E]\!]$ be an EXAM reachable state reachable. Then:*

1. Uniqueness. *There are no repeated names in $\mathbb{A}$.*
2. Freshness. *Different binding occurrences in s refer to different variable names.*
3. Bijection. *The set of names in $\mathbb{A}$ is in 1–1 correspondence with the set of names in P, that is, $\mathtt{names}(\mathbb{A}) = \mathtt{names}(P)$.*
4. Freeness. *The free variables of $\mathbb{A}$ are globally free, that is, $\mathtt{fv}(\mathbb{A}) \cap \mathtt{dom}\, E = \emptyset$.*
5. Local scope. *For every sub-term of the form $\lambda x.\, t$ in a job in $\mathtt{supp}(P)$ or in E, there are no occurrences of x outside of t. Moreover, in the environment $[x_1 \leftarrow t_1] : .. : [x_n \leftarrow t_n]$, there are no occurrences of x_i in t_j if $i \leq j$.*

The read-back of an EXAM state is defined as a multi-context, but for reachable states it is a term, as stated by Point 2 of the next lemma, proved by putting together the bijection invariant for reachable states (Lemma 3.3) and Point 1.

Lemma 4 (Reachable states read back to terms).

1. *Let $\mathbb{A}$ be an approximant and let X be a set of uniquely named jobs such that $\mathtt{names}(\mathbb{A}) \subseteq \mathtt{names}(X)$. Then $\underline{\mathbb{A}}_X$ is a term.*
2. *Let s be a reachable state. Then its read-back $\underline{s}$ is a term.*

Contextual Read-Back. The key point of the β-projection property is proving that the read-back of the data structures of a reachable state without the active term/job is an evaluation context—an external context in our case. This is ensured by the following lemma. It has a simple proof (using Lemma 4) because we can state it about approximants without mentioning reachable state, given that we know that the first component of a reachable state is always an approximant (because of Lemma 1). The lemma is then used in the proof of β-projection.

Lemma 5 (External context read-back). *Let X be a set of uniquely named jobs and $\mathbb{A}$ be an approximant with no repeated names such that $\mathtt{names}(\mathbb{A}) \setminus \mathtt{names}(X) = \{\alpha\}$. Then $\underline{\mathbb{A}}_X\langle\langle\cdot\rangle\rangle_\alpha$ is an external context.*

Theorem 3 (β-projection). *If $s \rightsquigarrow_\beta s'$ then $\underline{s} \to_{\mathtt{x}} \underline{s'}$.*

Now, we obtain the *runs to evaluations* part of the implementation theorem, which by the theorem about sufficient conditions for implementations (Theorem 1) follows from overhead transparency and β-projection.

Corollary 1 (EXAM runs to external evaluations). *For any EXAM run $\rho : t \triangleleft s \rightsquigarrow^*_{\mathtt{EXAM}} s'$ there exists a $\to_{\mathtt{x}}$-evaluation $e : t \to^*_{\mathtt{x}} \underline{s'}$. Moreover, $|e| = |\rho|_\beta$.*

Last, we analyze final states.

Proposition 5 (Characterization of final states). *Let s be a reachable final state. Then there is a normal form f such that $s = [\![f \mid \emptyset \mid E]\!]$ and $\underline{s} = f$. Moreover, if $s \equiv s'$ then s' is final and $\underline{s'} = f$.*

6.1 Leftmost Runs to Leftmost Evaluations

Now, we instantiate the EXAM with the stack template for pools, obtaining a machine implementing leftmost evaluation.

Leftmost EXAM. Let the *Leftmost EXAM* be the deterministic variant of the EXAM adopting the stack template for pools, that is, such that:

- Pools are lists $j_{\alpha_1} : .. : j_{\alpha_n}$ of named jobs, $\mathtt{new}(j_\alpha)$ creates the list containing only j_α, and the support of a pool is the set of jobs in the list;
- Selection pops from the pool, that is, if $P = j_{\alpha_1} : .. : j_{\alpha_n}$ then $j_\alpha \overset{\mathtt{sel}}{\leftarrow} P$ pops j_α from the list $j_\alpha : j_{\alpha_1} : .. : j_{\alpha_n}$;
- Both dropping and adding push on the list, and are inverses of selection.

Example 2. The Leftmost EXAM run on the same term $t := x(\mathtt{I}_y z)(\delta_w z)$ used for the Set EXAM in Example 1 follows (excluding the first three transitions, that are the same for both machines, as they are actually steps of the MAM).

Approx.	Pool	Env	Trans.	Selected Job
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$(\mathtt{I}_y z, \epsilon)_\beta : (\delta_w z, \epsilon)_\gamma$	ϵ	$\rightsquigarrow_{\mathtt{sea}_@}$	β
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$(\mathtt{I}_y, z)_\beta : (\delta_w z, \epsilon)_\gamma$	ϵ	$\rightsquigarrow_\beta$	β
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$(y, \epsilon)_\beta : (\delta_w z, \epsilon)_\gamma$	$[y{\leftarrow}z]$	$\rightsquigarrow_{\mathtt{sub}}$	β
$x\langle\cdot\rangle_\beta\langle\cdot\rangle_\gamma$	$(z, \epsilon)_\beta : (\delta_w z, \epsilon)_\gamma$	$[y{\leftarrow}z]$	$\rightsquigarrow_{\mathtt{sea}_\mathcal{V}}$	β
$xz\langle\cdot\rangle_\gamma$	$(\delta_w z, \epsilon)_\gamma$	$[y{\leftarrow}z]$	$\rightsquigarrow_{\mathtt{sea}_@}$	γ
$xz\langle\cdot\rangle_\gamma$	$(\delta_w, z)_\gamma$	$[y{\leftarrow}z]$	$\rightsquigarrow_\beta$	γ
$xz\langle\cdot\rangle_\gamma$	$(ww, \epsilon)_\gamma$	$[w{\leftarrow}z] : [y{\leftarrow}z]$	$\rightsquigarrow_{\mathtt{sea}_@}$	γ
$xz\langle\cdot\rangle_\gamma$	$(w, w)_\gamma$	$[w{\leftarrow}z] : [y{\leftarrow}z]$	$\rightsquigarrow_{\mathtt{sub}}$	γ
$xz\langle\cdot\rangle_\gamma$	$(z, w)_\gamma$	$[w{\leftarrow}z] : [y{\leftarrow}z]$	$\rightsquigarrow_{\mathtt{sea}_\mathcal{V}}$	γ
$xz(z\langle\cdot\rangle_{\gamma'})$	$(w, \epsilon)_{\gamma'}$	$[w{\leftarrow}z] : [y{\leftarrow}z]$	$\rightsquigarrow_{\mathtt{sub}}$	γ'
$xz(z\langle\cdot\rangle_{\gamma'})$	$(z, \epsilon)_{\gamma'}$	$[w{\leftarrow}z] : [y{\leftarrow}z]$	$\rightsquigarrow_{\mathtt{sea}_\mathcal{V}}$	γ'
$xz(zz)$	ϵ	$[w{\leftarrow}z] : [y{\leftarrow}z]$		

Proving that Leftmost EXAM runs read back to leftmost evaluations requires a new β-projection property. Its proof is based on the following quite complex invariant about instantiations of the approximants computed by the Leftmost EXAM. *Terminology*: a context C is *non-applying* if $C \neq D\langle\langle\cdot\rangle t\rangle$ for all D and t, that is, it does not apply the hole to an argument.

Proposition 6 (Leftmost context invariant). *Let*

- *$s = [\![\mathbb{A} \mid j_{\alpha_1} : .. : j_{\alpha_n} \mid E]\!]$ be a reachable Leftmost EXAM state with $n \geq 1$;*
- *f_j be a normal form for all j such that $1 \leq j < n$,*

- t_j *be a term for all* j *such that* $1 < j \leq n$*, and*
- L *be a non-applying leftmost context.*

Then $C^s_{f_1,..,f_{i-1}|L|t_{i+1},..,t_n} := \mathbb{A}\langle f_1\rangle_{\alpha_1}..\langle f_{i-1}\rangle_{\alpha_{i-1}}\langle L\rangle_{\alpha_i}\langle t_{i+1}\rangle_{\alpha_{i+1}}..\langle t_n\rangle_{\alpha_n}$ *is a non-applying leftmost context for every* $i \in \{1,\dots,n\}$*.*

From the invariant, it follows that a reachable state less the first job reads back to a leftmost context, which implies the leftmost variant of β-projection, in turn allowing us to project Leftmost EXAM runs on leftmost evaluations.

Lemma 6 (Leftmost context read-back). *Let* $s = [\![\mathbb{A} \mid j_{\alpha_1} : .. : j_{\alpha_n} \mid E]\!]$ *be a reachable Leftmost EXAM state with* $n \geq 1$ *and* $s_\bullet := [\![\mathbb{A} \mid j_{\alpha_2} : .. : j_{\alpha_n} \mid E]\!]$*. Then* $\underline{s_\bullet}\langle\langle\cdot\rangle\rangle_{\alpha_1}$ *is a non-applying leftmost context.*

Proposition 7 (Leftmost β-projection). *Let* s *be a reachable Leftmost EXAM state. If* $s \rightsquigarrow_\beta s'$ *then* $\underline{s} \to_{lo} \underline{s'}$*.*

Corollary 2 (Leftmost EXAM runs to leftmost evaluations). *For any Leftmost EXAM run* $\rho : t \triangleleft s \rightsquigarrow^*_{\texttt{EXAM}} s'$ *there exists a* $\to_{lo}$*-evaluation* $e : t \to^*_{lo} \underline{s'}$*. Moreover,* $|e| = |\rho|_\beta$*.*

7 Evaluations to Runs

Here, we develop the reflection of external evaluations to EXAM runs. By the abstract recipe for implementation theorems in Sect. 3, one needs to prove *overhead termination* and *β-reflection*. At the level of non-determinism, the external strategy is matched by the most permissive scheduling of jobs, that is, the set template. Therefore, we shall prove the result with respect to the Set EXAM.

Overhead Termination. To prove overhead termination, we define a measure. The measure does not depend on job names nor bound variable names, which is why the definition of the measure replaces them with underscores (and it is well defined even if it uses the renaming $E(x)^{\mathtt{R}}$).

Definition 8 (Overhead measure). *Let* j_α *be a job and* E *be an environment satisfying the freshness name property (together) of Lemma 3, and* s *be a reachable state. The overhead measures* $|j_\alpha, E|_\circ$ *and* $|s|_\circ$ *are defined as follows:*

$$\begin{array}{rll}
|(\lambda_.t, u : S)_, E|_\circ := 0 & & \\
|(\lambda_.t, \epsilon)_, E|_\circ := 1 + |(t, \epsilon)_, E|_\circ & & \\
|(tu, S)_, E|_\circ := 1 + |(t, u : S)_, E|_\circ & & \\
|(x, \epsilon)_, E|_\circ := 1 + |(E(x)^{\mathtt{R}}, \epsilon)_, E|_\circ & & \text{if } x \in \mathtt{dom}\, E \\
|(x, t_1 : .. : t_n)_, E|_\circ := 1 + \sum_{i=1}^n |(t_i, \epsilon)_, E|_\circ & \text{with } n \geq 0, & \text{if } x \notin \mathtt{dom}\, E \\
|[\![\mathbb{A} \mid P \mid E]\!]|_\circ := \sum_{j_\alpha \in \mathtt{supp}(P)} |j_\alpha, E|_\circ & &
\end{array}$$

Proposition 8 (Overhead termination). *Let* s *be a Set EXAM reachable state. Then* $s \rightsquigarrow_\circ^{|s|_\circ} s'$ *with* $s' \not\rightsquigarrow_\circ$*-normal.*

Addresses and β-Reflection. For the β-reflection property, we need a way to connect external redexes on terms with β-transitions on states. We use *addresses*.

Definition 9 (Address and sub-term at an address). *An* address $\mathtt{a}$ *is a string over the alphabet $\{l, r, \lambda\}$. The sub-term $t|_{\mathtt{a}}$ of a term t at address $\mathtt{a}$ is the following partial function (the last case of which means that in any case not covered by the previous ones $t|_{\mathtt{a}}$ is undefined):*

$$t|_{\epsilon} := t \qquad (tu)|_{l:\mathtt{a}} := t|_{\mathtt{a}} \qquad (tu)|_{r:\mathtt{a}} := u|_{\mathtt{a}} \qquad (\lambda x.t)|_{\lambda:\mathtt{a}} := t|_{\mathtt{a}}$$
$$\_|_{c:\mathtt{a}} := \bot \quad \text{if } c \in \{l, r, \lambda\}$$

The sub-term $\mathbb{C}|_{\mathtt{a}}$ at $\mathtt{a}$ of a multi-context is defined analogously. An address $\mathtt{a}$ is defined in t (resp. $\mathbb{C}$) if $t|_{\mathtt{a}} \neq \bot$ (resp. $\mathbb{C}|_{\mathtt{a}} \neq \bot$), and undefined otherwise.

There is a strong relationship between addresses in the approximant of a state and in the read-back of the state, as expressed by the following lemma. The lemma is then used to prove β-reflection, from which the *evaluation to runs* part of the implementation theorem follows.

Lemma 7. *Let $s = [\![\mathbb{A} \mid P \mid E]\!]$ be a state and $\mathtt{a}$ a defined address in $\mathbb{A}$. Then $\mathtt{a}$ is a defined address in $\underline{s}$, and $\underline{s}|_{\mathtt{a}}$ starts with the same constructor of $\mathbb{A}|_{\mathtt{a}}$ unless $\mathbb{A}|_{\mathtt{a}}$ is a named hole.*

Proposition 9 (β-reflection). *Let s be a $\rightsquigarrow_{\mathtt{o}}$-normal reachable state. If $\underline{s} \to_{\mathtt{x}} u$ then there exists s' such that $s \rightsquigarrow_{\beta} s'$ and $\underline{s'} = u$.*

Corollary 3 (Evaluations to runs). *For every $\to_{\mathtt{x}}$-evaluation $e : t \to_{\mathtt{x}}^{*} u$ there exists a Set EXAM run $\rho : t \triangleleft s \rightsquigarrow_{\mathtt{EXAM}}^{*} s'$ such that $\underline{s'} = u$.*

A similar result for leftmost evaluation and the Leftmost EXAM follows more easily from the characterization of final states (Proposition 5), overhead termination (Proposition 8), and determinism of the Leftmost EXAM—this is the standard pattern for deterministic strategies and machines, used for instance by Accattoli et al. for their machine for leftmost evaluation [4].

Names and Addresses. It is natural to wonder whether one can refine the EXAM by using addresses $\mathtt{a}$ as a more precise form of names for jobs. It is possible, it is enough to modify the EXAM as to extend at each step the name/address. For instance, transition $\rightsquigarrow_{\mathtt{sea}_@}$ would become:

Ap. | Pool | Env		Ap. | Pool | Env
$\mathbb{A} \mid (t\,u, S)_{\mathtt{a}} \overset{\mathtt{sel}}{\leftarrow} P \mid E$	$\rightsquigarrow_{\mathtt{sea}_@}$	$\mathbb{A} \mid (t, u : S)_{l:\mathtt{a}} \overset{\mathtt{dro}}{\rightarrow} P \mid E$

Then a β-transition of address $\mathtt{a}$ in a reachable state s corresponds exactly to a β-redex of address $\mathtt{a}$ in $\underline{s}$. We refrained from adopting addresses as names, however, because this is only useful for proving the β-reflection property of the EXAM, the machine does not need such an additional structure for its functioning.

8 Further Pool Templates

Least Level. Another sub-strategy of external reduction that computes β-normal forms is provided by *least level reduction* $\to_{\ell\ell}$, a notion from the linear logic literature. Picking a redex of minimal level, where the level is the number of arguments in which the redex is contained, is another predicate (similarly to the leftmost one) that ensures externality of an outermost redex. Note that the Ω redex in (1) (page 5) is not of minimal level (it has level 1 while the redex involving z has level 0). Least level reduction is non-deterministic but diamond. For instance the two redexes (of level 1) in $x(\mathtt{I}y)(\mathtt{I}z)$ are both least level. Note that the leftmost redex might not be least level, as in $x(x(\mathtt{I}y))(\mathtt{I}z)$, where the leftmost redex is $\mathtt{I}y$ and has level 2, while $\mathtt{I}z$ has level 1.

By replacing the stack template with a queue one, the Leftmost EXAM turns into a machine for least level evaluation. The key point is that when new jobs are created, which is done only by transition $\rightsquigarrow_{\mathtt{sea}_{\mathcal{V}}}$, they all have level $n+1$ where n is the level of the active job. To process jobs by increasing levels, then, it is enough to add the new jobs *at the end of the pool*, rather than at the beginning. This is an example where dropping (which pushes an element on top of the list of jobs) and adding (which adds at the end) are *not* the same operation.

Fair Template. Another interesting template is the one where pools are lists and dropping always adds at the end of the list. In this way the EXAM is *fair*, in the sense that even when it diverges, it keeps evaluating all jobs. This kind of strategies are of interest for infinitary λ-calculi, where one wants to compute all branches of an infinite normal form, instead of being stuck on one.

9 Conclusions

This paper studies two simple ideas and applies them to the paradigmatic case of strong call-by-name evaluation. Firstly, avoiding *backtracking on the search for redexes* by introducing *jobs* for each argument and jumping to the next job when one is finished. Secondly, modularizing the scheduling of jobs via a *pool interface* that can be instantiated by various concrete schedulers, called *pool templates*.

The outcome of the study is a compact, modular, and—we believe—elegant abstract machine for strong evaluation. In particular, we obtain the simplest machine for leftmost evaluation in the literature. Our study also gives a computational interpretation to the diamond non-determinism of strong call-by-name.

For the sake of simplicity, our study extends the MAM, which implements weak head reduction using global environments. Our technique, however, is reasonably modular in the underlying machine/notion of environment. One can, indeed, replace the MAM with Krivine abstract machine (KAM), which instead uses local environments, by changing only the fact that the jobs of the EXAM have to carry their own local environment. Similarly, the technique seems to be adaptable to the CEK or other machines for weak evaluation. It would be interesting to compare the outcome of these adaptations with existing machines for strong call-by-value [14,8,15] or strong call-by-need [12,16].

References

1. Accattoli, B.: The Useful MAM, a Reasonable Implementation of the Strong λ-Calculus. In: Väänänen, J.A., Hirvonen, Å., de Queiroz, R.J.G.B. (eds.) Logic, Language, Information, and Computation - 23rd International Workshop, WoLLIC 2016, Puebla, Mexico, August 16-19th, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9803, pp. 1–21. Springer (2016). https://doi.org/10.1007/978-3-662-52921-8_1
2. Accattoli, B., Barenbaum, P.: A diamond machine for strong evaluation. CoRR **abs/2309.12515** (2023), https://arxiv.org/abs/2309.12515
3. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling abstract machines. In: Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014. pp. 363–376 (2014). https://doi.org/10.1145/2628136.2628154
4. Accattoli, B., Barenbaum, P., Mazza, D.: A Strong Distillery. In: Programming Languages and Systems - 13th Asian Symposium, APLAS 2015. Lecture Notes in Computer Science, vol. 9458, pp. 231–250. Springer (2015). https://doi.org/10.1007/978-3-319-26529-2_13
5. Accattoli, B., Barras, B.: Environments and the complexity of abstract machines. In: Vanhoof, W., Pientka, B. (eds.) Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017. pp. 4–16. ACM (2017). https://doi.org/10.1145/3131851.3131855
6. Accattoli, B., Bonelli, E., Kesner, D., Lombardi, C.: A nonstandard standardization theorem. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 659–670. ACM (2014). https://doi.org/10.1145/2535838.2535886
7. Accattoli, B., Condoluci, A., Guerrieri, G., Sacerdoti Coen, C.: Crumbling abstract machines. In: Komendantskaya, E. (ed.) Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019. pp. 4:1–4:15. ACM (2019). https://doi.org/10.1145/3354166.3354169
8. Accattoli, B., Condoluci, A., Sacerdoti Coen, C.: Strong Call-by-Value is Reasonable, Implosively. In: 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021. pp. 1–14. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470630
9. Accattoli, B., Faggian, C., Guerrieri, G.: Factorization and normalization, essentially. In: Lin, A.W. (ed.) Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11893, pp. 159–180. Springer (2019). https://doi.org/10.1007/978-3-030-34175-6_9
10. Accattoli, B., Guerrieri, G.: Abstract machines for open call-by-value. Sci. Comput. Program. **184** (2019). https://doi.org/10.1016/j.scico.2019.03.002
11. Barendregt, H.P., Kennaway, R., Klop, J.W., Sleep, M.R.: Needed reduction and spine strategies for the lambda calculus. Inf. Comput. **75**(3), 191–231 (1987). https://doi.org/10.1016/0890-5401(87)90001-0
12. Biernacka, M., Biernacki, D., Charatonik, W., Drab, T.: An abstract machine for strong call by value. In: d. S. Oliveira, B.C. (ed.) Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 -

December 2, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12470, pp. 147–166. Springer (2020). https://doi.org/10.1007/978-3-030-64437-6_8
13. Biernacka, M., Biernacki, D., Lenglet, S., Schmitt, A.: Non-deterministic abstract machines. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland. LIPIcs, vol. 243, pp. 7:1–7:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.CONCUR.2022.7
14. Biernacka, M., Charatonik, W.: Deriving an abstract machine for strong call by need. In: Geuvers, H. (ed.) 4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany. LIPIcs, vol. 131, pp. 8:1–8:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.FSCD.2019.8
15. Biernacka, M., Charatonik, W., Drab, T.: A derived reasonable abstract machine for strong call by value. In: Veltri, N., Benton, N., Ghilezan, S. (eds.) PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021. pp. 6:1–6:14. ACM (2021). https://doi.org/10.1145/3479394.3479401
16. Biernacka, M., Charatonik, W., Drab, T.: A simple and efficient implementation of strong call by need by an abstract machine. Proc. ACM Program. Lang. **6**(ICFP), 109–136 (2022). https://doi.org/10.1145/3549822
17. Biernacka, M., Charatonik, W., Zielinska, K.: Generalized refocusing: From hybrid strategies to abstract machines. In: 2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK. pp. 10:1–10:17 (2017). https://doi.org/10.4230/LIPIcs.FSCD.2017.10
18. Boudol, G.: Computational semantics of term rewriting systems. In: Algebraic methods in semantics, pp. 169–236. Cambridge University Press (1986)
19. de Carvalho, D., Pagani, M., Tortora de Falco, L.: A semantic measure of the execution time in linear logic. Theor. Comput. Sci. **412**(20), 1884–1902 (2011). https://doi.org/10.1016/j.tcs.2010.12.017
20. Crégut, P.: Strongly reducing variants of the Krivine abstract machine. High. Order Symb. Comput. **20**(3), 209–230 (2007). https://doi.org/10.1007/s10990-007-9015-z
21. Dal Lago, U., Martini, S.: The weak lambda calculus as a reasonable machine. Theor. Comput. Sci. **398**(1-3), 32–50 (2008). https://doi.org/10.1016/j.tcs.2008.01.044
22. Danvy, O., Nielsen, L.R.: Refocusing in Reduction Semantics. Tech. Rep. RS-04-26, BRICS (2004)
23. García-Pérez, Á., Nogueira, P.: The full-reducing krivine abstract machine KN simulates pure normal-order reduction in lockstep: A proof via corresponding calculus. J. Funct. Program. **29**, e7 (2019). https://doi.org/10.1017/S0956796819000017
24. García-Pérez, Á., Nogueira, P., Moreno-Navarro, J.J.: Deriving the full-reducing Krivine machine from the small-step operational semantics of normal order. In: 15th International Symposium on Principles and Practice of Declarative Programming, PPDP'13. pp. 85–96. ACM (2013). https://doi.org/10.1145/2505879.2505887
25. Girard, J.: Light linear logic. Inf. Comput. **143**(2), 175–204 (1998). https://doi.org/10.1006/inco.1998.2700
26. Gonthier, G., Lévy, J.J., Melliès, P.A.: An abstract standardisation theorem. In: Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992. pp. 72–81. IEEE Computer Society (1992). https://doi.org/10.1109/LICS.1992.185521

27. Huet, G.P., Lévy, J.J.: Computations in orthogonal rewriting systems, I. In: Lassez, J., Plotkin, G.D. (eds.) Computational Logic - Essays in Honor of Alan Robinson. pp. 395–414. The MIT Press (1991)
28. Huet, G.P., Lévy, J.J.: Computations in orthogonal rewriting systems, II. In: Lassez, J., Plotkin, G.D. (eds.) Computational Logic - Essays in Honor of Alan Robinson. pp. 415–443. The MIT Press (1991)
29. Maranget, L.: Optimal derivations in weak lambda-calculi and in orthogonal terms rewriting systems. In: Wise, D.S. (ed.) Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991. pp. 255–269. ACM Press (1991). https://doi.org/10.1145/99583.99618
30. Melliès, P.A.: Description Abstraite de système de réécriture. PhD thesis, Paris 7 University (1996)
31. van Oostrom, V.: Normalisation in weakly orthogonal rewriting. In: Narendran, P., Rusinowitch, M. (eds.) Rewriting Techniques and Applications, 10th International Conference, RTA-99, Trento, Italy, July 2-4, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1631, pp. 60–74. Springer (1999). https://doi.org/10.1007/3-540-48685-2_5
32. Terese: Term rewriting systems., Cambridge tracts in theoretical computer science, vol. 55. Cambridge University Press (2003)