Implementing Open Call-by-Value

Beniamino Accattoli, Giulio Guerrieri

Implementing Open Call-by-Value

Beniamino Accattoli[1] and Giulio Guerrieri[2]

[1] INRIA, UMR 7161, LIX, École Polytechnique, beniamino.accattoli@inria.fr
[2] University of Oxford, Department of Computer Science, Oxford, United Kingdom, giulio.guerrieri@cs.ox.ac.uk

Abstract. The theory of the call-by-value λ-calculus relies on weak evaluation and closed terms, that are natural hypotheses in the study of programming languages. To model proof assistants, however, strong evaluation and open terms are required. Open call-by-value is the intermediate setting of weak evaluation with open terms, on top of which Grégoire and Leroy designed the abstract machine of Coq. This paper provides a theory of abstract machines for open call-by-value. The literature contains machines that are either simple but inefficient, as they have an exponential overhead, or efficient but heavy, as they rely on a labelling of environments and a technical optimization. We introduce a machine that is simple and efficient: it does not use labels and it implements open call-by-value within a bilinear overhead. Moreover, we provide a new fine understanding of how different optimizations impact on the complexity of the overhead.

This work is part of a wider research effort, the COCA HOLA project https://sites.google.com/site/beniaminoaccattoli/coca-hola.

1 Introduction

The λ-calculus is the computational model behind functional programming languages and proof assistants. A charming feature is that its definition is based on just one *macro-step* computational rule, *β-reduction*, and does not rest on any notion of machine or automaton. Compilers and proof assistants however are concrete tools that have to implement the λ-calculus in some way—a problem clearly arises. There is a huge gap between the abstract mathematical setting of the calculus and the technical intricacies of an actual implementation. This is why the issue is studied via intermediate *abstract machines*, that are implementation schemes with *micro-step* operations and without too many concrete details.

Closed and Strong λ-Calculus. Functional programming languages are based on a simplified form of λ-calculus, that we like to call *closed λ-calculus*, with two important restrictions. First, evaluation is *weak*, *i.e.* it does not evaluate function bodies. Second, terms are *closed*, that is, they have no free variables. The theory of the closed λ-calculus is much simpler than the general one.

Proof assistants based on the λ-calculus usually require the power of the full theory. Evaluation is then *strong*, *i.e.* unrestricted, and the distinction between

open and closed terms no longer makes sense, because evaluation has to deal with the issues of open terms even if terms are closed, when it enters function bodies. We refer to this setting as the *strong λ-calculus*.

Historically, the study of strong and closed λ-calculi have followed orthogonal approaches. Theoretical studies rather dealt with the strong λ-calculus, and it is only since the seminal work of Abramsky and Ong [1] that theoreticians started to take the closed case seriously. Dually, practical studies mostly ignored strong evaluation, with the notable exception of Crégut [13] (1990) and some very recent works [19,6,3]. Strong evaluation is nonetheless essential in the implementation of proof assistants or higher-order logic programming, typically for type-checking with dependent types as in the Edinburgh Logical Framework or the Calculus of Constructions, as well as for unification in simply typed frameworks like λ-prolog.

Open Call-by-Value. In a very recent work [8], we advocated the relevance of the *open λ-calculus*, a framework in between the closed and the strong ones, where evaluation is *weak* but terms may be *open*. Its key property is that the strong case can be described as the iteration of the open one into function bodies. The same cannot be done with the closed λ-calculus because—as already pointed out—entering into function bodies requires to deal with (locally) open terms.

The open λ-calculus did not emerge before because most theoretical studies focus on the *call-by-name* strong λ-calculus, and in call-by-name the distinction open/closed does not play an important role. Such a distinction, instead, is delicate for call-by-value evaluation, where Plotkin's original operational semantics [22] is not adequate for open terms. This issue is discussed at length in [8], where four extensions of Plotkin's semantics to open terms are compared and shown to be equivalent. That paper then introduces the expression *Open Call-by-Value* (shortened *Open CbV*) to refer to them as a whole, as well as *Closed CbV* and *Strong CbV* to concisely refer to the closed and strong call-by-value λ-calculus.

The Fireball Calculus. The simplest presentation of Open CbV is the *fireball calculus* λ_{fire}, obtained from the CbV λ-calculus by generalizing values into *fireballs*. Dynamically, β-redexes are allowed to fire only when the argument is a fireball (*fireball* is a pun on *fire-able*). The fireball calculus was introduced without a name by Paolini and Ronchi Della Rocca [21,23], then rediscovered independently first by Leroy and Grégoire [20], and then by Accattoli and Sacerdoti Coen [2]. Notably, on closed terms, λ_{fire} *coincides* with Plotkin's (Closed) CbV λ-calculus.

Coq by Levels. In [20] (2002) Leroy and Grégoire used the fireball calculus to improve the implementation of the Coq proof assistant. In fact, Coq rests on Strong CbV, but Leroy and Grégoire design an abstract machine for the fireball calculus (*i.e.* Open CbV) and then use it to evaluate Strong CbV *by levels*: the machine is first executed at top level (that is, out of all abstractions), and then re-launched recursively under abstractions. Their study is itself formalized in Coq, but it lacks an estimation of the efficiency of the machine.

In order to continue our story some basic facts about cost models and abstract machines have to be recalled (see [4] for a gentle tutorial).

Interlude 1: Size Explosion. It is well-known that λ-calculi suffer from a degeneracy called *size explosion*: there are families of terms whose size is linear in n, that evaluate in n β-steps, and whose result has size exponential in n. The problem is that the number of β-steps, the natural candidate as a time cost model, then seems not to be a reasonable cost model, because it does not even account for the time to write down the result of a computation—the *macro-step* character of β-reduction seems to forbid to count 1 for each step. This is a problem that affects all λ-calculi and all evaluation strategies.

Interlude 2: Reasonable Cost Models and Abstract Machines. Despite size explosion, surprisingly, the number of β-steps often *is* a reasonable cost model—so one can indeed count 1 for each β-step. There are no paradoxes: λ-calculi can be simulated in alternative formalisms employing some form of sharing, such as abstract machines. These settings manage compact representations of terms via *micro-step* operations and produce compact representations of the result, avoiding size explosion. Showing that a certain λ-calculus is reasonable usually is done by simulating it with a *reasonable* abstract machine, *i.e.* a machine implementable with overhead polynomial in the number of β-steps in the calculus. The design of a reasonable abstract machine depends very much on the kind of λ-calculus to be implemented, as different calculi admit different forms of size explosion and/or require more sophisticated forms of sharing. For strategies in the closed λ-calculus it is enough to use the ordinary technology for abstract machines, as first shown by Blelloch and Greiner [12], and then by Sands, Gustavsson, and Moran [24], and, with other techniques, by combining the results in Dal Lago and Martini's [15] and [14]. The case of the strong λ-calculus is subtler, and a more sophisticated form of sharing is necessary, as first shown by Accattoli and Dal Lago [7]. The topic of this paper is the study of reasonable machines for the intermediate case of Open CbV.

Fireballs are Reasonable. In [2] Accattoli and Sacerdoti Coen study Open CbV from the point of view of cost models. Their work provides 3 contributions:

1. *Open Size Explosion*: they show that Open CbV is subtler than Closed CbV by exhibiting a form of size explosions that is not possible in Closed CbV, making Open CbV closer to Strong CbV rather than to Closed CbV;
2. *Fireballs are Reasonable*: they show that the number of β-steps in the fireball calculus is nonetheless a reasonable cost model by exhibiting a reasonable abstract machine, called GLAMOUr, improving over Leroy and Grégoire's machine in [20] (see the conclusions for more on their machine);
3. *And Even Efficient*: they optimize the GLAMOUr into the Unchaining GLAMOUr, whose overhead is bilinear (*i.e.* linear in the number of β-steps *and* the size of the initial term), that is the best possible overhead.

This Paper. Here we present two machines, the Easy GLAMOUr and the Fast GLAMOUr, that are proved to be correct implementations of Open CbV and to have a polynomial and bilinear overhead, respectively. Their study refines the results of [2] along three axes:

1. *Simpler Machines*: both the GLAMOUr and the Unchaining GLAMOUr of [2] are sophisticated machines resting on a labeling of terms. The unchaining optimizations of the second machine is also quite heavy. Both the Easy GLAMOUr and the Fast GLAMOUr, instead, do not need labels and the Fast GLAMOUr is bilinear with no need of the unchaining optimization.
2. *Simpler Analyses*: the correctness and complexity analyses of the (Unchaining) GLAMOUr are developed in [2] via an informative but complex decomposition via explicit substitutions, by means of the distillation methodology [5]. Here, instead, we decode the Easy and Fast GLAMOUr directly to the fireball calculus, that turns out to be much simpler. Moreover, the complexity analysis of the Fast GLAMOUr, surprisingly, turns out to be straightforward.
3. *Modular Decomposition of the Overhead*: we provide a fine analysis of how different optimizations impact on the complexity of the overhead of abstract machines for Open CbV. In particular, it turns out that one of the optimizations considered essential in [2], namely *substituting abstractions on-demand*, is not mandatory for reasonable machines—the Easy GLAMOUr does not implement it and yet it is reasonable. We show, however, that this is true only as long as one stays *inside* Open CbV because the optimization is instead mandatory for Strong CbV (seen by Grégoire and Leroy as Open CbV *by levels*). To our knowledge substituting abstractions on-demand is an optimization introduced in [7] and currently no proof assistant implements it. Said differently, our work shows that the technology currently in use in proof assistants is, at least theoretically, unreasonable.

Summing up, this paper does not improve the known bound on the overhead of abstract machines for Open CbV, as the one obtained in [2] is already optimal. Its contributions instead are a simplification and a finer understanding of the subtleties of implementing Open CbV: we introduce simpler abstract machines whose complexity analyses are elementary and carry a new modular view of how different optimizations impact on the complexity of the overhead.

In particular, while [2] shows that Open CbV is subtler than Closed CbV, here we show that Open CbV is simpler than Strong CbV, and that defining Strong CbV as iterated Open CbV, as done by Grégoire and Leroy in [20], may introduce an explosion of the overhead, if done naively.

A longer version of this paper is available on Arxiv [9]. It contains two Appendices, one with a glossary of rewriting theory and one with omitted proofs.

2 The Fireball Calculus λ_{fire} & Open Size Explosion

In this section we introduce the fireball calculus, the presentation of Open CbV we work with in this paper, and show the example of size explosion peculiar to the open setting. Alternative presentations of Open CbV can be found in [8].

The Fireball Calculus. The fireball calculus λ_{fire} is defined in Fig. 1. The idea is that the values of the call-by-value λ-calculus, given by abstractions and

$$
\begin{array}{cc}
\text{Terms} & t,u,s,r ::= x \mid \lambda x.t \mid tu \\
\text{Fireballs} & f,f',f'' ::= \lambda x.t \mid i \\
\text{Inert Terms} & i,i',i'' ::= x f_1 \dots f_n \quad n \geq 0 \\
\text{Evaluation Contexts} & E ::= \langle\cdot\rangle \mid tE \mid Et
\end{array}
$$

$$
\begin{array}{cc}
\textsc{Rule at Top Level} & \textsc{Contextual closure} \\
(\lambda x.t)(\lambda y.u) \mapsto_{\beta_\lambda} t\{x \leftarrow \lambda y.u\} & E\langle t\rangle \to_{\beta_\lambda} E\langle u\rangle \quad \text{if } t \mapsto_{\beta_\lambda} u \\
(\lambda x.t)i \mapsto_{\beta_i} t\{x \leftarrow i\} & E\langle t\rangle \to_{\beta_i} E\langle u\rangle \quad \text{if } t \mapsto_{\beta_i} u
\end{array}
$$

$$
\text{Reduction} \qquad \to_{\beta_f} := \to_{\beta_\lambda} \cup \to_{\beta_i}
$$

Fig. 1. The Fireball Calculus λ_{fire}

variables, are generalized to fireballs, by extending variables to more general *inert terms*. Actually fireballs and inert terms are defined by mutual induction (in Fig. 1). For instance, $\lambda x.y$ is a fireball as an abstraction, while x, $y(\lambda x.x)$, xy, and $(z(\lambda x.x))(zz)(\lambda y.(zy))$ are fireballs as inert terms.

The main feature of inert terms is that they are open, normal, and that when plugged in a context they cannot create a redex, hence the name (they are not so-called *neutral terms* because they might have β-redexes under abstractions). In Grégoire and Leroy's presentation [20], inert terms are called *accumulators* and fireballs are simply called values.

Terms are always identified up to α-equivalence and the set of free variables of a term t is denoted by $\mathtt{fv}(t)$. We use $t\{x \leftarrow u\}$ for the term obtained by the capture-avoiding substitution of u for each free occurrence of x in t.

Evaluation is given by *call-by-fireball* β-reduction $\to_{\beta_f}$: the β-rule can fire, *lighting up* the argument, only when it is a fireball (*fireball* is a catchier version of *fire-able term*). We actually distinguish two sub-rules: one that *lights up* abstractions, noted $\to_{\beta_\lambda}$, and one that *lights up* inert terms, noted $\to_{\beta_i}$ (see Fig. 1). Note that evaluation is weak (*i.e.* it does not reduce under abstractions).

Properties of the Calculus. A famous key property of Closed CbV (whose evaluation is exactly $\to_{\beta_\lambda}$) is *harmony*: given a closed term t, either it diverges or it evaluates to an abstraction, *i.e.* t is β_λ-normal iff t is an abstraction. The fireball calculus satisfies an analogous property in the *open* setting by replacing abstractions with fireballs (Prop. 1.1). Moreover, the fireball calculus is a conservative extension of Closed CbV: on closed terms it collapses on Closed CbV (Prop. 1.2). No other presentation of Open CbV has these properties.

Proposition 1 (Distinctive Properties of λ_{fire}). *Let t be a term.*

1. Open Harmony*: t is β_f-normal iff t is a fireball.*
2. Conservative Open Extension*: $t \to_{\beta_f} u$ iff $t \to_{\beta_\lambda} u$ whenever t is closed.*

The rewriting rules of λ_{fire} have also many good operational properties, studied in [8] and summarized in the following proposition.

Proposition 2 (Operational Properties of λ_{fire}, [8]). *The reduction $\to_{\beta_f}$ is strongly confluent, and all β_f-normalizing derivations d (if any) from a term*

t have the same length $|d|_{\beta_f}$, the same number $|d|_{\beta_\lambda}$ of β_λ-steps, and the same number $|d|_{\beta_i}$ of β_i-steps.

Right-to-Left Evaluation. As expected from a *calculus*, the evaluation rule $\to_{\beta_f}$ of λ_{fire} is *non-deterministic*, because in the case of an application there is no fixed order in the evaluation of the left and right subterms. Abstract machines however implement *deterministic* strategies. We then fix a deterministic strategy (which fires β_f-redexes from right to left and is the one implemented by the machines of the next sections). By Prop. 2, the choice of the strategy does not impact on existence of a result, nor on the result itself or on the number of steps to reach it. It does impact however on the design of the machine, which selects β_f-redexes from right to left.

The *right-to-left evaluation strategy* $\to_{\mathtt{r}\beta_f}$ is defined by closing the root rules $\mapsto_{\beta_\lambda}$ and $\mapsto_{\beta_i}$ in Fig. 1 by *right contexts*, a special kind of evaluation contexts defined by $R ::= \langle\cdot\rangle \mid tR \mid Rf$. The next lemma ensures our definition is correct.

Lemma 3 (Properties of $\to_{\mathtt{r}\beta_f}$). *Let t be a term.*

1. Completeness*: t has $\to_{\beta_f}$-redex iff t has a $\to_{\mathtt{r}\beta_f}$-redex.*
2. Determinism*: t has at most one $\to_{\mathtt{r}\beta_f}$-redex.*

Example 4. Let $t := (\lambda z.z(yz))\lambda x.x$. Then, $t \to_{\mathtt{r}\beta_f} (\lambda x.x)(y\,\lambda x.x) \to_{\mathtt{r}\beta_f} y\,\lambda x.x$, where the final term $y\,\lambda x.x$ is a fireball (and β_f-normal).

Open Size Explosion. Fireballs are delicate, they easily *explode*. The simplest instance of *open size explosion* (not existing in Closed CbV) is a variation over the famous looping term $\Omega := (\lambda x.xx)(\lambda x.xx) \to_{\beta_\lambda} \Omega \to_{\beta_\lambda} \dots$. In Ω there is an infinite sequence of duplications. In the size exploding family there is a sequence of n nested duplications. We define two families, the family $\{t_n\}_{n\in\mathbb{N}}$ of size exploding terms and the family $\{i_n\}_{n\in\mathbb{N}}$ of results of evaluating $\{t_n\}_{n\in\mathbb{N}}$:

$$t_0 := y \qquad t_{n+1} := (\lambda x.xx)t_n \qquad\qquad i_0 := y \qquad i_{n+1} := i_n i_n$$

We use $|t|$ for the size of a term, *i.e.* the number of symbols to write it.

Proposition 5 (Open Size Explosion, [2]). *Let $n \in \mathbb{N}$. Then $t_n \to_{\beta_i}^n i_n$, moreover $|t_n| = O(n)$, $|i_n| = \Omega(2^n)$, and i_n is an inert term.*

Circumventing Open Size Explosion. Abstract machines implementing the substitution of inert terms, such as the one described by Grégoire and Leroy in [20] are unreasonable because for the term t_n of the size exploding family they compute the full result i_n. The machines of the next sections are reasonable because they avoid the substitution of inert terms, that is justified by the following lemma.

Lemma 6 (Inert Substitutions Can Be Avoided). *Let t, u be terms and i be an inert term. Then, $t \to_{\beta_f} u$ iff $t\{x\leftarrow i\} \to_{\beta_f} u\{x\leftarrow i\}$.*

Lemma 6 states that the substitution of an inert term cannot create redexes, which is why it can be avoided. For general terms, only direction $\Rightarrow$ holds, because substitution can create redexes, as in $(xy)\{x\leftarrow\lambda z.z\} = (\lambda z.z)y$. Direction $\Leftarrow$, instead, is distinctive of inert terms, of which it justifies the name.

3 Preliminaries on Abstract Machines, Implementations, and Complexity Analyses

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

Implementations. For every machine one has to prove that it correctly implements the strategy in the λ-calculus it was conceived for. Our notion, tuned towards complexity analyses, requires a perfect match between the number of β-steps of the strategy and the number of β-transitions of the machine execution.

Definition 7 (Machine Implementation). *A machine* $\mathtt{M}$ implements a strategy $\to$ *on* λ*-terms via a decoding* $\underline{\cdot}$ *when given a* λ*-term* t *the following holds:*

1. Executions to Derivations*: for any* $\mathtt{M}$*-execution* $\rho\colon t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s$ *there exists a* $\to$*-derivation* $d\colon t \to^{*} \underline{s}$.
2. Derivations to Executions*: for every* $\to$*-derivation* $d\colon t \to^{*} u$ *there exists a* $\mathtt{M}$*-execution* $\rho\colon t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s$ *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* $|d|$ *of the derivation* d*, i.e.* $|d| = |\rho|_{\beta}$.

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

Definition 8 (Implementation System). *A machine* $\mathtt{M}$*, a strategy* $\to$*, and a decoding* $\underline{\cdot}$ *form an* implementation system *if the following conditions hold:*

1. β-Projection*:* $s \rightsquigarrow_\beta s'$ *implies* $\underline{s} \to \underline{s'}$*;*
2. Overhead Transparency*:* $s \rightsquigarrow_{\mathtt{o}} s'$ *implies* $\underline{s} = \underline{s'}$*;*
3. Overhead Transitions Terminate*:* $\rightsquigarrow_{\mathtt{o}}$ *terminates;*
4. Determinism*: both* $\mathtt{M}$ *and* $\to$ *are deterministic;*
5. Progress*:* $\mathtt{M}$ *final states decode to* $\to$*-normal terms.*

Theorem 9 (Sufficient Condition for Implementations). *Let* $(\mathtt{M}, \to, \underline{\cdot})$ *be an* implementation system*. Then,* $\mathtt{M}$ *implements* $\to$ *via* $\underline{\cdot}$*.*

The proof of Thm. 9 is a clean and abstract generalization of the concrete reasoning already at work in [5,2,3,4].

Parameters for Complexity Analyses. By the *derivations-to-executions* part of the implementation (Point 2 in Def. 7), given a derivation $d\colon t_0 \to^n u$ there is a shortest execution $\rho\colon t_0^\circ \rightsquigarrow_{\mathtt{M}}^* s$ such that $\underline{s} = u$. Determining *the complexity of a machine* $\mathtt{M}$ amounts to bound the complexity of a concrete implementation of ρ on a RAM model, as a function of two fundamental parameters:

1. *Input*: the size $|t_0|$ of the initial term t_0 of the derivation d;
2. *β-Steps/Transitions*: the length $n = |d|$ of the derivation d, that coincides with the number $|\rho|_\beta$ of β-transitions in ρ by the β-matching requirement for implementations (Point 3 in Def. 7).

A machine is *reasonable* if its complexity is polynomial in $|t_0|$ and $|\rho|_\beta$, and it is *efficient* if it is linear in both parameters. So, a strategy is reasonable (resp. efficient) if there is a reasonable (resp. efficient) machine implementing it. In Sect. 4-5 we study a reasonable machine implementing right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ in λ_{fire}, thus showing that it is a reasonable strategy. In Sect. 6 we optimize the machine to make it efficient. By Prop. 2, this implies that *every* strategy in λ_{fire} is efficient.

Recipe for Complexity Analyses. For complexity analyses on a machine $\mathtt{M}$, overhead transitions $\rightsquigarrow_{\mathtt{o}}$ are further separated into two classes:

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

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

1. *Number of Transitions*: bounding the length of the execution ρ, by bounding the number of overhead transitions. This part splits into two subparts:
   i. *Substitution vs β*: bounding the number $|\rho|_{\mathtt{s}}$ of substitution transitions in ρ using the number of β-transitions;
   ii. *Commutative vs Substitution*: bounding the number $|\rho|_{\mathtt{c}}$ of substitution transitions in ρ using the size of the input and $|\rho|_{\mathtt{s}}$; the latter—by the previous point—induces a bound with respect to β-transitions.
2. *Cost of Single Transitions*: bounding the cost of concretely implementing a single transition of $\mathtt{M}$. Here it is usually necessary to go beyond the abstract

$$\begin{array}{l|l|l}
\phi ::= \lambda x.\overline{u}@\epsilon \mid x@\pi \quad E ::= \epsilon \mid [x{\leftarrow}\phi]:E & \underline{\epsilon} \coloneqq \langle\cdot\rangle & t{\downarrow_\epsilon} \coloneqq t \quad t{\downarrow_{[x\leftarrow\phi]:E}} \coloneqq t\{x{\leftarrow}\underline{\phi}\}{\downarrow_E} \\
\pi ::= \epsilon \mid \phi:\pi \quad\quad s ::= (D,\overline{t},\pi,E) & \underline{\phi:\pi} \coloneqq \langle\langle\cdot\rangle\underline{\phi}\rangle\underline{\pi} & C_s \coloneqq \underline{D}\langle\underline{\pi}\rangle{\downarrow_E} \\
D ::= \epsilon \mid D:\overline{t}\lozenge\pi & \underline{t@\pi} \coloneqq \langle t\rangle\underline{\pi} & \underline{s} \coloneqq \underline{D}\langle\langle\overline{t}\rangle\underline{\pi}\rangle{\downarrow_E} = C_s\langle\overline{t}{\downarrow_E}\rangle \\
 & \underline{D:\overline{t}\lozenge\pi} \coloneqq \underline{D}\langle\langle\overline{t}\langle\cdot\rangle\rangle\underline{\pi}\rangle & \text{where } s=(D,\overline{t},\pi,E)
\end{array}$$

Dump	Code	Stack	Global Env		Dump	Code	Stack	Global Env
D	$\overline{t}\overline{u}$	π	E	$\rightsquigarrow_{\mathtt{c}_1}$	$D:\overline{t}\lozenge\pi$	$\overline{u}$	ϵ	E
$D:\overline{t}\lozenge\pi$	$\lambda x.\overline{u}$	ϵ	E	$\rightsquigarrow_{\mathtt{c}_2}$	D	$\overline{t}$	$\lambda x.\overline{u}@\epsilon:\pi$	E
$D:\overline{t}\lozenge\pi$	x	π'	E	$\rightsquigarrow_{\mathtt{c}_3}$	D	$\overline{t}$	$x@\pi':\pi$	E
							if $E(x)=\bot$ or $E(x)=y@\pi''$	
D	$\lambda x.\overline{t}$	$\phi:\pi$	E	$\rightsquigarrow_\beta$	D	$\overline{t}$	π	$[x{\leftarrow}\phi]E$
D	x	π	$E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2$	$\rightsquigarrow_{\mathtt{s}}$	D	$(\lambda y.\overline{u})^\alpha$	π	$E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2$

where $(\lambda y.\overline{u})^\alpha$ is any well-named code α-equivalent to $\lambda y.\overline{u}$ such that its bound names are fresh with respect to those in D, π and $E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2$.

Fig. 2. Easy GLAMOUr machine: data-structures (stacks π, dumps D, global env. E, states s), unfolding $t{\downarrow_E}$, decoding $\underline{\cdot}$ (stacks are decoded to contexts in postfix notation for plugging, *i.e.* we write $\langle\overline{t}\rangle\underline{\pi}$ rather than $\underline{\pi}\langle\overline{t}\rangle$), and transitions.

level, making some (high-level) assumption on how codes and data-structure are concretely represented. Commutative transitions are designed on purpose to have constant cost. Each substitution transition has a cost linear in the size of the initial term thanks to an invariant (to be proved) ensuring that only subterms of the initial term are duplicated and substituted along an execution. Each β-transition has a cost either constant or linear in the input.

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

(Linear) Logical Reading. Let us mention that our partitioning of transitions into β, substitution, and commutative ones admits a proof-theoretical view, as machine transitions can be seen as cut-elimination steps [11,5]. Commutative transitions correspond to commutative cases, while β and substitution are principal cases. Moreover, in linear logic the β transition corresponds to the multiplicative case while the substitution transition to the exponential one. See [5] for more details.

4 Easy GLAMOUr

In this section we introduce the Easy GLAMOUr, a simplified version of the GLAMOUr machine from [2]: unlike the latter, the Easy GLAMOUr does not need any labeling of codes to provide a reasonable implementation.

With respect to the literature on abstract machines for CbV, our machines are unusual in two respects. First, and more importantly, they use a single *global* environment instead of *closures* and *local environments*. Global environments are used in a minority of works [17,24,16,5,2,6,3] and induce simpler, more abstract machines where α-equivalence is pushed to the meta-level (in the operation $\overline{t}^\alpha$ in

$\rightsquigarrow_{\mathtt{s}}$ in Fig. 2-3). This on-the-fly α-renaming is harmless with respect to complexity analyses, see also discussions in [5,4]. Second, argument stacks contain pairs of a code and a stack, to implement some of the machine transitions in constant time.

Background. GLAMOUr stands for *Useful* (*i.e.* optimized to be *reasonable*) *Open* (reducing open terms) *Global* (using a single global environment) LAM, and LAM stands for *Leroy Abstract Machine*, an ordinary machine implementing right-to-left Closed CbV, defined in [5]. In [2] the study of the GLAMOUr was done according to the distillation approach of [5], *i.e.* by decoding the machine towards a λ-calculus with explicit substitutions. Here we do not follow the distillation approach, we decode directly to λ_{fire}, which is simpler.

Machine Components. The Easy GLAMOUr is defined in Fig. 2. A machine state s is a quadruple $(D, \bar{t}, \pi, E)$ given by:

- *Code* $\bar{t}$: a term not considered up to α-equivalence, which is why it is over-lined;
- *Argument Stack* π: it contains the arguments of the current code. Note that stacks items ϕ are pairs $x@\pi$ and $\lambda x.\overline{u}@\epsilon$. These pairs allow to implement some of the transitions in constant time. The pair $x@\pi$ codes the term $\langle x\rangle\underline{\pi}$ (defined in Fig. 2—the decoding is explained below) that would be obtained by putting x in the context obtained by decoding the argument stack π. The pair $\lambda x.\overline{u}@\epsilon$ is used to inject abstractions into pairs, so that items ϕ can be uniformly seen as pairs $\bar{t}@\pi$ of a code $\bar{t}$ and a stack π.
- *Dump* D: a second stack, that together with the argument stack π is used to walk through the code and search for the next redex to reduce. The dump is extended with an entry $\bar{t}\Diamond\pi$ every time evaluation enters in the right subterm $\overline{u}$ of an application $\bar{t}\overline{u}$. The entry saves the left part $\bar{t}$ of the application and the current stack π, to restore them when the evaluation of the right subterm $\overline{u}$ is over. The dump D and the stack π decode to an evaluation context.
- *Global Environment* E: a list of explicit (*i.e.* delayed) substitutions storing substitutions generated by the redexes encountered so far. It is used to implement micro-step evaluation (*i.e.* the substitution for one variable occurrence at a time). We write $E(x) = \bot$ if in E there are no entries of the form $[x{\leftarrow}\phi]$. Often $[x{\leftarrow}\phi]E$ stands for $[x{\leftarrow}\phi]:E$.

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

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

– *Commutative Transitions* $\rightsquigarrow_{\mathtt{c}}$: they locate and expose the next redex according to the right-to-left strategy, by rearranging the data-structures. They are invisible on the calculus. The commutative rule $\rightsquigarrow_{\mathtt{c}_1}$ forces evaluation to be right-to-left on applications: the machine processes first the right subterm $\overline{u}$, saving the left sub-term $\overline{t}$ on the dump together with its current stack π. The role of $\rightsquigarrow_{\mathtt{c}_2}$ and $\rightsquigarrow_{\mathtt{c}_3}$ is to backtrack to the entry on top of the dump. When the right subterm, *i.e.* the pair $\overline{t}@\pi$ of current code and stack, is finally in normal form, it is pushed on the stack and the machine backtracks.

O for Open: note condition $E(x) = \bot$ in $\rightsquigarrow_{\mathtt{c}_3}$—that is how the Easy GLAMOUr handles open terms. *U for Useful*: note condition $E(x) = y@\pi''$ in $\rightsquigarrow_{\mathtt{c}_3}$—inert terms are never substituted, according to Lemma 6. Removing the useful side-condition one recovers Grégoire and Leroy's machine [20]. Note that terms substituted by $\rightsquigarrow_{\mathtt{s}}$ are always abstractions and never variables—this fact will play a role in Sect. 6. *Garbage Collection*: it is here simply ignored, or, more precisely, it is encapsulated at the meta-level, in the decoding function. It is well-known that this is harmless for the study of time complexity.

Compiling, Decoding and Invariants. A term t is compiled to the machine *initial state* $t^\circ = (\epsilon, \overline{t}, \epsilon, \epsilon)$, where $\overline{t}$ is a well-named term α-equivalent to t. Conversely, every machine state s decodes to a term $\underline{s}$ (see the top right part of Fig. 2), having the shape $C_s\langle \overline{t}\downarrow_E\rangle$, where $\overline{t}\downarrow_E$ is a λ-term, obtained by applying to the code the meta-level substitution $\downarrow_E$ induced by the global environment E, and C_s is an evaluation context, obtained by decoding the stack π and the dump D and then applying $\downarrow_E$. Note that, to improve readability, stacks are decoded to contexts in postfix notation for plugging, *i.e.* we write $\langle \overline{t}\rangle\underline{\pi}$ rather than $\underline{\pi}\langle\overline{t}\rangle$ because π is a context that puts arguments in front of $\overline{t}$.

Example 10. To have a glimpse of how the Easy GLAMOUr works, let us show how it implements the derivation $t \coloneqq (\lambda z.z(yz))\lambda x.x \to^2_{\mathtt{r}\beta_f} y\,\lambda x.x$ of Ex. 4:

Dump	Code	Stack	Global Environment	
ϵ	$(\lambda z.z(yz))\lambda x.x$	ϵ	ϵ	$\rightsquigarrow_{\mathtt{c}_1}$
$\lambda z.z(yz)\diamondsuit\epsilon$	$\lambda x.x$	ϵ	ϵ	$\rightsquigarrow_{\mathtt{c}_2}$
ϵ	$\lambda z.z(yz)$	$\lambda x.x@\epsilon$	ϵ	$\rightsquigarrow_{\beta}$
ϵ	$z(yz)$	ϵ	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{c}_1}$
$z\diamondsuit\epsilon$	yz	ϵ	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{c}_1}$
$z\diamondsuit\epsilon : y\diamondsuit\epsilon$	z	ϵ	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{s}}$
$z\diamondsuit\epsilon : y\diamondsuit\epsilon$	$\lambda x'.x'$	ϵ	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{c}_2}$
$z\diamondsuit\epsilon$	y	$\lambda x'.x'@\epsilon$	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{c}_3}$
ϵ	z	$y@(\lambda x'.x'@\epsilon)$	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{s}}$
ϵ	$\lambda x''.x''$	$y@(\lambda x'.x'@\epsilon)$	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\beta}$
ϵ	x''	ϵ	$[x''\leftarrow y@(\lambda x'.x'@\epsilon)] : [z\leftarrow\lambda x.x@\epsilon]$	

Note that the initial state is the compilation of the term t, the final state decodes to the term $y\,\lambda x.x$, and the two β-transitions in the execution correspond to the two $\to_{\mathtt{r}\beta_f}$-steps in the derivation considered in Ex. 4.

The study of the Easy GLAMOUr machine relies on the following invariants.

Lemma 11 (Easy GLAMOUr Qualitative Invariants). *Let $s = (D, \bar{t}, \pi, E)$ be a reachable state. Then:*

1. Name:
   1. Explicit Substitution*: if $E = E'[x \leftarrow \overline{u}]E''$ then x is fresh wrt $\overline{u}$ and E'';*
   2. Abstraction*: if $\lambda x.\overline{u}$ is a subterm of D, $\bar{t}$, π, or E, x may occur only in $\overline{u}$;*
2. Fireball Item*: $\underline{\phi}$ and $\underline{\phi}\downarrow_E$ are inert terms if $\phi = x@\pi'$, and abstractions otherwise, for every item ϕ in π, in E, and in every stack in D;*
3. Contextual Decoding*: $C_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context.*

Implementation Theorem. The invariants are used to prove the implementation theorem by proving that the hypotheses of Thm. 9 hold, that is, that the Easy GLAMOUr, $\to_{\mathtt{r}\beta_f}$ and $\underline{\cdot}$ form an implementation system.

Theorem 12 (Easy GLAMOUr Implementation). *The Easy GLAMOUr implements right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ in λ_{fire} (via the decoding $\underline{\cdot}$).*

5 Complexity Analysis of the Easy GLAMOUr

The analysis of the Easy GLAMOUr is done according to the recipe given at the end of Sect. 3. The result (see Thm. 17 below) is that the Easy GLAMOUr is linear in the number $|\rho|_\beta$ of β-steps/transitions and quadratic in the size $|t_0|$ of the initial term t_0, *i.e.* its overhead has complexity $O((1 + |\rho|_\beta) \cdot |t_0|^2)$.

The analysis relies on a quantitative invariant, the crucial *subterm invariant*, ensuring that $\rightsquigarrow_{\mathtt{s}}$ duplicates only subterms of the initial term, so that the cost of duplications is connected to one of the two parameters for complexity analyses.

Lemma 13 (Subterm Invariant). *Let $\rho\colon t_0^\circ \rightsquigarrow^* (D, \bar{t}, \pi, E)$ be an Easy GLAMOUr execution. Every subterm $\lambda x.\overline{u}$ of D, $\bar{t}$, π, or E is a subterm of t_0.*

Intuition About Complexity Bounds. The number $|\rho|_{\mathtt{s}}$ of substitution transitions $\rightsquigarrow_{\mathtt{s}}$ depends on both parameters for complexity analyses, the number $|\rho|_\beta$ of β-transitions *and* the size $|t_0|$ of the initial term. Dependency on $|\rho|_\beta$ is standard, and appears in every machine [12,24,5,2,6,3]—sometimes it is quadratic, here it is linear, in Sect. 6 we come back to this point. Dependency on $|t_0|$ is also always present, but usually only for *the cost* of a single $\rightsquigarrow_{\mathtt{s}}$ transition, since only subterms of t_0 are duplicated, as ensured by the subterm invariant. For the Easy GLAMOUr, instead, also *the number* of $\rightsquigarrow_{\mathtt{s}}$ transitions depends—linearly—on $|t_0|$: this is a side-effect of dealing with open terms. Since both the cost and the number of $\rightsquigarrow_{\mathtt{s}}$ transitions depend on $|t_0|$, the dependency is quadratic.

The following family of terms shows the dependency on $|t_0|$ in isolation (*i.e.*, with no dependency on $|\rho|_\beta$). Let $r_n \coloneqq \lambda x.(\dots((y \underbrace{x)x)\dots)x}_{n}$ and consider:

$$u_n \coloneqq r_n r_n = (\lambda x.(\dots((y \overbrace{x)x)\dots)x}^{n})r_n \to_{\beta_\lambda} (\dots((y \overbrace{r_n)r_n)\dots)r_n}^{n}. \qquad (1)$$

Forgetting about commutative transitions, the Easy GLAMOUr would evaluate u_n with one β-transition $\rightsquigarrow_\beta$ and n substitution transitions $\rightsquigarrow_{\mathsf{s}}$, each one duplicating r_n, whose size (as well as the size of the initial term u_n) is linear in n.

The number $|\rho|_{\mathsf{c}}$ of commutative transitions $\rightsquigarrow_{\mathsf{c}}$, roughly, is linear in the amount of code involved in the evaluation process. This amount is given by the initial code plus the code produced by duplications, that is bounded by the number of substitution transitions times the size of the initial term. The number of commutative transitions is then $O((1+|\rho|_\beta)\cdot|t_0|^2)$. Since each one has constant cost, this is also a bound to their cost.

Number of Transitions 1: Substitution vs β Transitions. The number $|\rho|_{\mathsf{s}}$ of substitution transitions is proven (see Cor. 15 below) to be bilinear, *i.e.* linear in $|t_0|$ and $|\rho|_\beta$, by means of a measure.

The *free size* $|\cdot|_{\mathsf{free}}$ of a code counts the number of free variable occurrences that are not under abstractions. It is defined and extended to states as follows:

$$\begin{array}{rr} |x|_{\mathsf{free}} \coloneqq 1 & |\epsilon|_{\mathsf{free}} \coloneqq 0 \\ |\lambda y.\overline{u}|_{\mathsf{free}} \coloneqq 0 & |\phi : \pi|_{\mathsf{free}} \coloneqq |\phi|_{\mathsf{free}} + |\pi|_{\mathsf{free}} \\ |\overline{t}\overline{u}|_{\mathsf{free}} \coloneqq |t|_{\mathsf{free}} + |u|_{\mathsf{free}} & |D : (\overline{t}, \pi)|_{\mathsf{free}} \coloneqq |t|_{\mathsf{free}} + |\pi|_{\mathsf{free}} + |D|_{\mathsf{free}} \end{array}$$

$$|(D, \overline{t}, \pi, E)|_{\mathsf{free}} \coloneqq |D|_{\mathsf{free}} + |\overline{t}|_{\mathsf{free}} + |\pi|_{\mathsf{free}}.$$

Lemma 14 (Free Occurrences Invariant). *Let $\rho\colon t_0^\circ \rightsquigarrow^* s$ be an Easy GLAMOUr execution. Then, $|s|_{\mathsf{free}} \leq |t_0|_{\mathsf{free}} + |t_0| \cdot |\rho|_\beta - |\rho|_{\mathsf{s}}$.*

Corollary 15 (Bilinear Number of Substitution Transitions). *Let $\rho : t_0^\circ \rightsquigarrow^* s$ be an Easy GLAMOUr execution. Then, $|\rho|_{\mathsf{s}} \leq (1 + |\rho|_\beta) \cdot |t_0|$.*

Number of Transitions 2: Commutative vs Substitution Transitions. The bound on the number $|\rho|_{\mathsf{c}}$ of commutative transitions is found by means of a (different) measure on states. The bound is linear in $|t_0|$ and in $|\rho|_{\mathsf{s}}$, which means—by applying the result just obtained in Cor. 15—*quadratic* in $|t_0|$ and linear in $|\rho|_\beta$.

The *commutative size* of a state is defined as $|(D, \overline{t}, \pi, E)|_{\mathsf{c}} \coloneqq |\overline{t}| + \Sigma_{\overline{u} \Diamond \pi' \in D} |\overline{u}|$, where $|\overline{t}|$ is the usual size of codes.

Lemma 16 (Number of Commutative Transitions). *Let $\rho\colon t_0^\circ \rightsquigarrow^* s$ be an Easy GLAMOUr execution. Then $|\rho|_{\mathsf{c}} \leq |\rho|_{\mathsf{c}} + |s|_{\mathsf{c}} \leq (1 + |\rho|_{\mathsf{s}}) \cdot |t_0| \in O((1 + |\rho|_\beta) \cdot |t_0|^2)$.*

Cost of Single Transitions. We need to make some hypotheses on how the Easy GLAMOUr is going to be itself implemented on RAM:

1. *Variable (Occurrences) and Environment Entries*: a variable is a memory location, a variable occurrence is a reference to it, and an environment entry $[x \leftarrow \phi]$ is the fact that the location associated to x contains ϕ.
2. *Random Access to Global Environments*: the environment E can be accessed in $O(1)$ (in $\rightsquigarrow_{\mathsf{s}}$) by just following the reference given by the variable occurrence under evaluation, with no need to access E sequentially, thus ignoring its list structure (used only to ease the definition of the decoding).

Dump	Code	Stack	Global Env		Dump	Code	Stack	Global Env
D	$\overline{t}\,\overline{u}$	π	E	$\rightsquigarrow_{\mathtt{c}_1}$	$D : \overline{t} \lozenge \pi$	$\overline{u}$	ϵ	E
$D : \overline{t} \lozenge \pi$	$\lambda x.\overline{u}$	ϵ	E	$\rightsquigarrow_{\mathtt{c}_2}$	D	$\overline{t}$	$\lambda x.\overline{u}@\epsilon : \pi$	E
$D : \overline{t} \lozenge \pi$	x	π'	E	$\rightsquigarrow_{\mathtt{c}_3}$	D	$\overline{t}$	$x@\pi' : \pi$	E
			if $E(x) = \bot$ or $E(x) = y@\pi''$ or $(E(x) = \lambda y.\overline{u}@\epsilon$ and $\pi' = \epsilon)$					
D	$\lambda x.\overline{t}$	$y@\epsilon : \pi$	E	$\rightsquigarrow_{\beta_1}$	D	$\overline{t}\{x \leftarrow y\}$	π	E
D	$\lambda x.\overline{t}$	$\phi : \pi$	E	$\rightsquigarrow_{\beta_2}$	D	$\overline{t}$	π	$[x \leftarrow \phi]E$
								if $\phi \neq y@\epsilon$
D	x	$\phi : \pi$	$E_1[x \leftarrow \lambda y.\overline{u}@\epsilon]E_2$	$\rightsquigarrow_{\mathtt{s}}$	D	$(\lambda y.\overline{u})^{\alpha}$	$\phi : \pi$	$E_1[x \leftarrow \lambda y.\overline{u}@\epsilon]E_2$

Fig. 3. Fast GLAMOUr (data-structures, decoding, and $(\lambda y.\overline{u})^{\alpha}$ defined as in Fig. 2).

With these hypotheses it is clear that β and overhead transitions can be implemented in $O(1)$. The substitution transition $\rightsquigarrow_{\mathtt{s}}$ needs to copy a code from the environment (the renaming $\overline{t}^{\alpha}$) and can be implemented in $O(|t_0|)$, as the subterm to copy is a subterm of t_0 by the subterm invariant (Lemma 13) and the environment can be accessed in $O(1)$.

Summing Up. By putting together the bounds on the number of transitions with the cost of single transitions we obtain the overhead of the machine.

Theorem 17 (Easy GLAMOUr Overhead Bound). *Let $\rho\colon t_0^{\circ} \rightsquigarrow^* s$ be an Easy GLAMOUr execution. Then ρ is implementable on RAM in $O((1 + |\rho|_{\beta}) \cdot |t_0|^2)$, i.e. linear in the number of β-transitions (aka the length of the derivation $d\colon t_0 \rightarrow^*_{\mathtt{r}\beta_f} \underline{s}$ implemented by ρ) and quadratic in the size of the initial term t_0.*

6 Fast GLAMOUr

In this section we optimize the Easy GLAMOUr, obtaining a machine, the Fast GLAMOUr, whose dependency from the size of the initial term is linear, instead of quadratic, providing a bilinear—thus optimal—overhead (see Thm. 21 below and compare it with Thm. 17 on the Easy GLAMOUr). We invite the reader to go back to equation (1) at page 12, where the quadratic dependency was explained. Note that in that example the substitutions of r_n do not create β_f-redexes, and so they are useless. The Fast GLAMOUr avoids these useless substitutions and it implements the example with no substitutions at all.

Optimization: Abstractions On-Demand. The difference between the Easy GLAMOUr and the machines in [2] is that, whenever the former encounters a variable occurrence x bound to an abstraction $\lambda y.\overline{t}$ in the environment, it replaces x with $\lambda y.\overline{t}$, while the latter are more parsimonious. They implement an optimization that we call *substituting abstractions on-demand*: x is replaced by $\lambda y.\overline{t}$ only if this is useful to obtain a β-redex, that is, only if the argument stack is non-empty. The Fast GLAMOUr, defined in Fig. 3, upgrades the Easy GLAMOUr with *substitutions of abstractions on-demand*—note the new side-condition for $\rightsquigarrow_{\mathtt{c}_3}$ and the non-empty stack in $\rightsquigarrow_{\mathtt{s}}$.

Abstractions On-Demand and the Substitution of Variables. The new optimization however has a consequence. To explain it, let us recall the role of another optimization, *no substitution of variables*. In the Easy GLAMOUr, abstractions are at depth 1 in the environment: there cannot be chains of renamings, *i.e.* of substitutions of variables for variable, ending in abstractions (so, there cannot be chains like $[x{\leftarrow}y@\epsilon][y{\leftarrow}z@\epsilon][z{\leftarrow}\lambda z'.\bar{t}@\epsilon]$). This property implies that the overhead is linear in $|\rho|_\beta$ and it is induced by the fact that variables cannot be substituted. If variables can be substituted then the overhead becomes quadratic in $|\rho|_\beta$—this is what happens in the GLAMOUr machine in [2]. The relationship between *substituting variables* and a linear/quadratic overhead is studied in-depth in [10].

Now, because the Fast GLAMOUr substitutes abstractions on-demand, variable occurrences that are not applied are not substituted by abstractions. The question becomes what to do when the code is an abstraction $\lambda x.\bar{t}$ and the top of the stack argument ϕ is a simple variable occurrence $\phi = y@\epsilon$ (potentially bound to an abstraction in the environment E) because if one admits that $[x{\leftarrow}y@\epsilon]$ is added to E then the depth of abstractions in the environment may be arbitrary and so the dependency on $|\rho|_\beta$ may be quadratic, as in the GLAMOUr. There are two possible solutions to this issue. The complex one, given by the Unchaining GLAMOUr in [2], is to add labels and a further unchaining optimization. The simple one is to split the β-transition in two, handling this situation with a new rule that renames x as y in the code $\bar{t}$ without touching the environment—this exactly what the Fast GLAMOUr does with $\rightsquigarrow_{\beta_1}$ and $\rightsquigarrow_{\beta_2}$. The consequence is that abstractions stay at depth 1 in E, and so the overhead is indeed bilinear.

The simple solution is taken from Sands, Gustavsson, and Moran's [24], where they use it on a call-by-name machine. Actually, it repeatedly appears in the literature on abstract machines often with reference to space consumption and *space leaks*, for instance in Wand's [26], Friedman et al.'s [18], and Sestoft's [25].

Fast GLAMOUr. The machine is in Fig. 3. Its data-structures, compiling and decoding are exactly as for the Easy GLAMOUr.

Example 18. Let us now show how the derivation $t := (\lambda z.z(yz))\lambda x.x \to^2_{\mathtt{r}\beta_f} y\,\lambda x.x$ of Ex. 4 is implemented by the Fast GLAMOUr. The execution is similar to that of the Easy GLAMOUr in Ex. 10, since they implement the same derivation and hence have the same initial state. In particular, the first five transitions in the Fast GLAMOUr (omitted here) are the same as in the Easy GLAMOUr (see Ex. 10 and replace $\rightsquigarrow_\beta$ with $\rightsquigarrow_{\beta_2}$). Then, the Fast GLAMOUr executes:

Dump	Code	Stack	Global Environment	
$z\lozenge\epsilon : y\lozenge\epsilon$	z	ϵ	$[z{\leftarrow}\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{c}_3}$
$z\lozenge\epsilon$	y	$z@\epsilon$	$[z{\leftarrow}\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{c}_3}$
ϵ	z	$y@(z@\epsilon)$	$[z{\leftarrow}\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{s}}$
ϵ	$\lambda x''.x''$	$y@(z@\epsilon)$	$[z{\leftarrow}\lambda x.x@\epsilon]$	$\rightsquigarrow_{\beta_2}$
ϵ	x''	ϵ	$[x''{\leftarrow}y@(z@\epsilon)] : [z{\leftarrow}\lambda x.x@\epsilon]$	

The Fast GLAMOUr executes only one substitution transition (the Easy GLAMOUr takes two) since the replacement of z with $\lambda x.x$ from the environment is *on-demand* (*i.e.* useful to obtain a β-redex) only for the first occurrence of z in $z(yz)$.

The Fast GLAMOUr satisfies the same invariants (the qualitative ones—the *fireball item* is slightly different—as well as the subterm one, see [9]) and also forms an implementation system with respect to $\to_{\mathtt{r}\beta_f}$ and $\underline{\cdot}$. Therefore,

Theorem 19 (Fast GLAMOUr Implementation). *The Fast GLAMOUr implements right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ in λ_{fire} (via the decoding $\underline{\cdot}$).*

Complexity Analysis. What changes is the complexity analysis, that, surprisingly, is simpler. First, we focus on *the number* of overhead transitions. The *substitution vs β transitions* part is simply trivial. Note that a substitution transition $\rightsquigarrow_{\mathtt{s}}$ is always immediately followed by a β-transition, because substitutions are done only *on-demand*—therefore, $|\rho|_{\mathtt{s}} \leq |\rho|_\beta + 1$. It is easy to remove the $+1$: executions must have a $\rightsquigarrow_{\beta_2}$ transition before any substitution one, otherwise the environment is empty and no substitutions are possible—thus $|\rho|_{\mathtt{s}} \leq |\rho|_\beta$.

For the *commutative vs substitution transitions* the exact same measure and the same reasoning of the Easy GLAMOUr provide the same bound, namely $|\rho|_{\mathtt{c}} \leq (1+|\rho|_{\mathtt{s}}) \cdot |t_0|$. What improves is the dependency of the commutatives from β-transitions (obtained by substituting the bound for substitution transitions), that is now linear because so is that of substitutions—so, $|\rho|_{\mathtt{c}} \leq (1 + |\rho|_\beta) \cdot |t_0|$.

Lemma 20 (Number of Overhead Transitions). *Let $\rho\colon t_0^\circ \rightsquigarrow^* s$ be a Fast GLAMOUr execution. Then,*

1. Substitution *vs* β Transitions*:* $|\rho|_{\mathtt{s}} \leq |\rho|_\beta$.
2. Commutative *vs* Substitution Transitions*:* $|\rho|_{\mathtt{c}} \leq (1+|\rho|_{\mathtt{s}}) \cdot |t_0| \leq (1+|\rho|_\beta) \cdot |t_0|$.

Cost of Single Transitions and Global Overhead. For the cost of single transitions, note that $\rightsquigarrow_{\mathtt{c}}$ and $\rightsquigarrow_{\beta_2}$ have (evidently) cost $O(1)$ while $\rightsquigarrow_{\mathtt{s}}$ and $\rightsquigarrow_{\beta_1}$ have cost $O(|t_0|)$ by the subterm invariant. Then we can conclude with

Theorem 21 (Fast GLAMOUr Bilinear Overhead). *Let $\rho\colon t_0^\circ \rightsquigarrow^* s$ be a Fast GLAMOUr execution. Then ρ is implementable on RAM in $O((1 + |\rho|_\beta) \cdot |t_0|)$, i.e. linear in the number of β-transitions (aka the length of the derivation $d\colon t_0 \to^*_{\mathtt{r}\beta_f} \underline{s}$ implemented by ρ) and the size of the initial term.*

7 Conclusions

Modular Overhead. The overhead of implementing Open CbV is measured with respect to the size $|t_0|$ of the initial term and the number n of β-steps. We showed that its complexity depends crucially on three choices about substitution.

The first is whether to substitute inert terms that are not variables. If they are substituted, as in Grégoire and Leroy's machine [20], then the overhead is exponential in $|t_0|$ because of open size explosion (Prop. 5) and the implementation is then unreasonable. If they are not substituted, as in the machines studied here and in [2], then the overhead is polynomial.

The other two parameters are whether to substitute variables, and whether abstractions are substituted whenever or only *on-demand*, and they give rise to the following table of machines and reasonable overheads:

	Sub of Abs Whenever	*Sub of Abs On-Demand*
Sub of Variables	Slow GLAMOUr $O((1+n^2)\cdot\|t_0\|^2)$	GLAMOUr $O((1+n^2)\cdot\|t_0\|)$
No Sub of Variables	Easy GLAMOUr $O((1+n)\cdot\|t_0\|^2)$	Fast / Unchaining GLAMOUr $O((1+n)\cdot\|t_0\|)$

The Slow GLAMOUr has been omitted for lack of space, because it is slow and involved, as it requires the labeling mechanism of the (Unchaining) GLAMOUr developed in [2]. It is somewhat surprising that the Fast GLAMOUr presented here has the best overhead and it is also the easiest to analyze.

Abstractions On-Demand: Open CbV is simpler than Strong CbV. We explained that Grégoire and Leroy's machine for Coq as described in [20] is unreasonable. Its actual implementation, on the contrary, does not substitute non-variable inert terms, so it is reasonable for Open CbV. None of the versions, however, substitutes abstractions on-demand (nor, to our knowledge, does any other implementation), despite the fact that it is a necessary optimization in order to have a reasonable implementation of Strong CbV, as we now show. Consider the following size exploding family (obtained by applying s_n to the identity $I \coloneqq \lambda x.x$), from [4]:

$$s_1 \coloneqq \lambda x.\lambda y.(yxx) \quad s_{n+1} \coloneqq \lambda x.(s_n(\lambda y.(yxx))) \qquad r_0 \coloneqq I \quad r_{n+1} \coloneqq \lambda y.(y r_n r_n)$$

Proposition 22 (Abstraction Size Explosion). *Let $n>0$. Then $s_n I \to_{\beta\lambda}^{n} r_n$. Moreover, $|s_n I| = O(n)$, $|r_n| = \Omega(2^n)$, $s_n I$ is closed, and r_n is normal.*

The evaluation of $s_n I$ produces 2^n non-applied copies of I (in r_n), so a strong evaluator not substituting abstractions on-demand must have an exponential overhead. Note that evaluation is weak but the 2^n copies of I are substituted under abstraction: this is why machines for Closed and Open CbV can be reasonable without substituting abstractions on-demand.

The Danger of Iterating Open CbV Naively. The size exploding example in Prop. 22 also shows that iterating reasonable machines for Open CbV is subtle, as it may induce unreasonable machines for Strong CbV, if done naively. Evaluating Strong CbV by iterating the Easy GLAMOUr (that does not substitute abstractions on-demand), indeed, induces an exponential overhead, while iterating the Fast GLAMOUr provides an efficient implementation.

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

References

1. Abramsky, S., Ong, C.L.: Full Abstraction in the Lazy Lambda Calculus. Inf. Comput. 105(2), 159–267 (1993)
2. Accattoli, B., Sacerdoti Coen, C.: On the Relative Usefulness of Fireballs. In: LICS 2015. pp. 141–155 (2015)

3. Accattoli, B.: The Useful MAM, a Reasonable Implementation of the Strong λ-Calculus. In: WoLLIC 2016. pp. 1–21 (2016)
4. Accattoli, B.: The Complexity of Abstract Machines. In: WPTE 2016 (invited paper). pp. 1–15 (2017)
5. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling abstract machines. In: ICFP 2014. pp. 363–376 (2014)
6. Accattoli, B., Barenbaum, P., Mazza, D.: A Strong Distillery. In: APLAS 2015. pp. 231–250 (2015)
7. Accattoli, B., Dal Lago, U.: Beta Reduction is Invariant, Indeed. In: CSL-LICS 2014. pp. 8:1–8:10 (2014)
8. Accattoli, B., Guerrieri, G.: Open Call-by-Value. In: APLAS 2016. pp. 206–226 (2016)
9. Accattoli, B., Guerrieri, G.: Implementing Open Call-by-Value (Extended Version). CoRR abs/1701.08186 (2017), `https://arxiv.org/abs/1701.08186`
10. Accattoli, B., Sacerdoti Coen, C.: On the Value of Variables. In: WoLLIC 2014. pp. 36–50 (2014)
11. Ariola, Z.M., Bohannon, A., Sabry, A.: Sequent calculi and abstract machines. ACM Trans. Program. Lang. Syst. 31(4) (2009)
12. Blelloch, G.E., Greiner, J.: A Provable Time and Space Efficient Implementation of NESL. In: ICFP '96. pp. 213–225 (1996)
13. Crégut, P.: An Abstract Machine for Lambda-Terms Normalization. In: LISP and Functional Programming. pp. 333–340 (1990)
14. Dal Lago, U., Martini, S.: Derivational complexity is an invariant cost model. In: FOPARA 2009. pp. 100–113 (2009)
15. Dal Lago, U., Martini, S.: On constructor rewrite systems and the lambda-calculus. In: ICALP 2009. pp. 163–174 (2009)
16. Danvy, O., Zerny, I.: A synthetic operational account of call-by-need evaluation. In: PPDP. pp. 97–108 (2013)
17. Fernández, M., Siafakas, N.: New Developments in Environment Machines. Electr. Notes Theor. Comput. Sci. 237, 57–73 (2009)
18. Friedman, D.P., Ghuloum, A., Siek, J.G., Winebarger, O.L.: Improving the lazy Krivine machine. Higher-Order and Symbolic Computation 20(3), 271–293 (2007)
19. García-Pérez, Á., Nogueira, P., Moreno-Navarro, J.J.: Deriving the full-reducing Krivine machine from the small-step operational semantics of normal order. In: PPDP. pp. 85–96 (2013)
20. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: ICFP '02. pp. 235–246 (2002)
21. Paolini, L., Ronchi Della Rocca, S.: Call-by-value Solvability. ITA 33(6), 507–534 (1999)
22. Plotkin, G.D.: Call-by-Name, Call-by-Value and the lambda-Calculus. Theor. Comput. Sci. 1(2), 125–159 (1975)
23. Ronchi Della Rocca, S., Paolini, L.: The Parametric λ-Calculus – A Metamodel for Computation. Springer (2004)
24. Sands, D., Gustavsson, J., Moran, A.: Lambda Calculi and Linear Speedups. In: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. pp. 60–84 (2002)
25. Sestoft, P.: Deriving a Lazy Abstract Machine. J. Funct. Program. 7(3), 231–264 (1997)
26. Wand, M.: On the correctness of the Krivine machine. Higher-Order and Symbolic Computation 20(3), 231–235 (2007)