Implementing Open Call-by-Value (Extended Version)

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.

This is a longer version of a paper accepted to FSEN 2017. 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].

Terms	$t, u, s, r ::= x \mid \lambda x.t \mid tu$
Fireballs	$f, f', f'' ::= \lambda x.t \mid i$
Inert Terms	$i, i', i'' ::= x f_1 \dots f_n \quad n \geq 0$
Evaluation Contexts	$E ::= \langle\cdot\rangle \mid tE \mid Et$
Rule at Top Level	Contextual closure
$(\lambda x.t)(\lambda y.u) \mapsto_{\beta_\lambda} t\{x{\leftarrow}\lambda y.u\}$	$E\langle t\rangle \rightarrow_{\beta_\lambda} E\langle u\rangle \quad \text{if } t \mapsto_{\beta_\lambda} u$
$(\lambda x.t)i \mapsto_{\beta_i} t\{x{\leftarrow}i\}$	$E\langle t\rangle \rightarrow_{\beta_i} E\langle u\rangle \quad \text{if } t \mapsto_{\beta_i} u$
Reduction	$\rightarrow_{\beta_f} := \rightarrow_{\beta_\lambda} \cup \rightarrow_{\beta_i}$

Fig. 1. The Fireball Calculus λ_{fire}

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 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 $\rightarrow_{\beta_f}$: 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 $\rightarrow_{\beta_\lambda}$, and one that *lights up* inert terms, noted $\rightarrow_{\beta_i}$ (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 $\rightarrow_{\beta_\lambda}$) 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.* Proof p. 21
1. Open Harmony*: t is β_f-normal iff t is a fireball.*
2. Conservative Open Extension*: $t \rightarrow_{\beta_f} u$ iff $t \rightarrow_{\beta_\lambda} u$ 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.* Proof p. 22

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.

Proof p. 22 **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 $t \coloneqq (\lambda z.z(yz))\lambda x.x$. 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 $\Omega \coloneqq (\lambda x.xx)(\lambda x.xx) \to_{\beta_\lambda} \Omega \to_{\beta_\lambda} \dots$. 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 $\{t_n\}_{n\in\mathbb{N}}$:

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

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

Proof p. 23 **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.

Proof p. 24 **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*:* $s \leadsto_{\beta} s'$ *implies* $\underline{s} \to \underline{s'}$*;*
2. Overhead Transparency*:* $s \leadsto_{\mathsf{o}} s'$ *implies* $\underline{s} = \underline{s'}$*;*
3. Overhead Transitions Terminate*:* $\leadsto_{\mathsf{o}}$ *terminates;*
4. Determinism*: both* $\mathtt{M}$ *and* $\to$ *are deterministic;*
5. Progress*:* $\mathtt{M}$ *final states decode to* $\to$*-normal terms.*

Proof p. 26 **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 $\rho \colon t_0^{\circ} \leadsto_{\mathtt{M}}^{*} s$ 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 $\leadsto_{\mathsf{o}}$ are further separated into two classes:

1. *Substitution Transitions* $\leadsto_{\mathsf{s}}$: they are in charge of the substitution process;
2. *Commutative Transitions* $\leadsto_{\mathsf{c}}$: 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:
    1. *Substitution vs β*: bounding the number $|\rho|_{\mathsf{s}}$ of substitution transitions in ρ using the number of β-transitions;

$$\begin{array}{ll|rlll}
\phi ::= \lambda x.\overline{u}@\epsilon \mid x@\pi & E ::= \epsilon \mid [x{\leftarrow}\phi]:E & \underline{\epsilon} & := \langle\cdot\rangle & t{\downarrow}_\epsilon := t & t{\downarrow}_{[x\leftarrow\phi]:E} := t\{x{\leftarrow}\underline{\phi}\}{\downarrow}_E \\
\pi ::= \epsilon \mid \phi : \pi & s ::= (D, \overline{t}, \pi, E) & \underline{\phi : \pi} & := \langle\langle\cdot\rangle\underline{\phi}\rangle\underline{\pi} & C_s := \underline{D}\langle\underline{\pi}\rangle{\downarrow}_E & \\
D ::= \epsilon \mid D : \overline{t}\Diamond\pi & & \underline{t@\pi} & := \langle t\rangle\underline{\pi} & \underline{s} := \underline{D}\langle\langle\overline{t}\rangle\underline{\pi}\rangle{\downarrow}_E = C_s\langle\overline{t}{\downarrow}_E\rangle & \\
 & & \underline{D:\overline{t}\Diamond\pi} & := \underline{D}\langle\langle\overline{t}\langle\cdot\rangle\rangle\underline{\pi}\rangle & \text{where } s = (D, \overline{t}, \pi, E) &
\end{array}$$

Dump	Code	Stack	Global Env		Dump	Code	Stack	Global Env
D	$\overline{t}\overline{u}$	π	E	$\rightsquigarrow_{\mathtt{c}_1}$	$D:\overline{t}\Diamond\pi$	$\overline{u}$	ϵ	E
$D:\overline{t}\Diamond\pi$	$\lambda x.\overline{u}$	ϵ	E	$\rightsquigarrow_{\mathtt{c}_2}$	D	$\overline{t}$	$\lambda x.\overline{u}@\epsilon : \pi$	E
$D:\overline{t}\Diamond\pi$	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.

   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 `M`. Here it is usually necessary to go beyond the abstract 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 [10,5]. 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 $(D, \overline{t}, \pi, E)$ given by:

- *Code* $\overline{t}$: 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 $\overline{t}@\pi$ of a code $\overline{t}$ 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 $\overline{t}\lozenge\pi$ every time evaluation enters in the right subterm $\overline{u}$ of an application $\overline{t}\overline{u}$. The entry saves the left part $\overline{t}$ 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 $(\lambda x.\overline{t})\phi$, except that it puts a new delayed substitution $[x{\leftarrow}\phi]$ in the environment instead of doing the meta-level substitution $\overline{t}\{x{\leftarrow}\phi\}$ of the argument in the body of the abstraction;

– *Substitution Transition* $\rightsquigarrow_{\mathsf{s}}$: 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 $\rightsquigarrow_{\mathsf{s}}$ cannot be observed at the coarser granularity of λ_{fire}.
– *Commutative Transitions* $\rightsquigarrow_{\mathsf{c}}$: 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 $\rightsquigarrow_{\mathsf{c}_1}$ 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 $\rightsquigarrow_{\mathsf{c}_2}$ and $\rightsquigarrow_{\mathsf{c}_3}$ 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 $\rightsquigarrow_{\mathsf{c}_3}$—that is how the Easy GLAMOUr handles open terms. *U for Useful*: note condition $E(x) = y@\pi''$ in $\rightsquigarrow_{\mathsf{c}_3}$—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 $\rightsquigarrow_{\mathsf{s}}$ 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 $t \coloneqq (\lambda z.z(yz))\lambda x.x \rightarrow^2_{\mathtt{r}\beta_f} y\,\lambda x.x$ of Ex. 4:

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

Proof p. 26

Lemma 11 (Easy GLAMOUr Qualitative Invariants). *Let $s = (D, \overline{t}, \pi, E)$ 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, $\overline{t}$, π, or E, x may occur only in $\overline{u}$;*
3. 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;*
4. 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.

Proof p. 30

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.

Proof p. 30

Lemma 13 (Subterm Invariant). *Let $\rho\colon t_0^\circ \rightsquigarrow^* (D, \overline{t}, \pi, E)$ be an Easy GLAMOUr execution. Every subterm $\lambda x.\overline{u}$ of D, $\overline{t}$, 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 $r_n \coloneqq \lambda x.(\underbrace{\ldots((y\,x)x)\ldots)x}_{n}$ and consider:

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

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:

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

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

Lemma 14 (Free Occurrences Invariant). *Let ρ: $t_0^\circ \rightsquigarrow^* s$ be an Easy GLAMOUr execution. Then, $|s|_{\mathsf{free}} \leq |t_0|_{\mathsf{free}} + |t_0| \cdot |\rho|_\beta - |\rho|_{\mathsf{s}}$.* Proof p. 31

Corollary 15 (Bilinear Number of Substitution Transitions). *Let ρ : $t_0^\circ \rightsquigarrow^* s$ be an Easy GLAMOUr execution. Then, $|\rho|_{\mathsf{s}} \leq (1 + |\rho|_\beta) \cdot |t_0|$.* Proof p. 31

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 ρ: $t_0^\circ \rightsquigarrow^* s$ be an Easy GLAMOUr execution. Then $|\rho|_{\mathsf{c}} \leq |\rho|_{\mathsf{c}} + |s|_{\mathsf{c}} \leq (1 + |\rho|_{\mathsf{s}}) \cdot |t_0| \in O((1 + |\rho|_\beta) \cdot |t_0|^2)$.* Proof p. 32

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 $\leadsto_{\mathtt{s}}$) 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).

With these hypotheses it is clear that β and overhead transitions can be implemented in $O(1)$. The substitution transition $\leadsto_{\mathtt{s}}$ needs to copy a code from the environment (the renaming $\bar{t}^{\alpha}$) 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.

Proof p. 32

Theorem 17 (Easy GLAMOUr Overhead Bound). *Let $\rho\colon t_0^{\circ} \leadsto^* s$ 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 $d\colon t_0 \to_{\mathtt{r}\beta_f}^* \underline{s}$ 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 13, 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 $\lambda y.\bar{t}$ in the environment, it replaces x with $\lambda y.\bar{t}$, while the latter are more parsimonious. They implement an optimization that we call *substituting abstractions on-demand*: x is replaced by $\lambda y.\bar{t}$ 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 $\leadsto_{\mathtt{c}_3}$ and the non-empty stack in $\leadsto_{\mathtt{s}}$.

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).

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 $[x{\leftarrow}y@\epsilon][y{\leftarrow}z@\epsilon][z{\leftarrow}\lambda z'.\overline{t}@\epsilon]$). 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 [9].

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 $\lambda x.\overline{t}$ 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 $\overline{t}$ 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	
$z\Diamond\epsilon : y\Diamond\epsilon$	z	ϵ	$[z{\leftarrow}\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{c}_3}$
$z\Diamond\epsilon$	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 Appendix B.5) and also forms an implementation system with respect to $\rightarrow_{\mathtt{r}\beta_f}$ and $\underline{\cdot}$. Therefore,

Proof p. 37

Theorem 19 (Fast GLAMOUr Implementation). *The Fast GLAMOUr implements right-to-left evaluation $\rightarrow_{\mathtt{r}\beta_f}$ 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, $|\rho|_{\mathtt{c}} \leq (1 + |\rho|_{\beta}) \cdot |t_0|$.

Proof p. 37

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

Proof p. 38

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 $d \colon t_0 \rightarrow^{*}_{\mathtt{r}\beta_f} \underline{s}$ 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]:

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

Proposition 22 (Abstraction Size Explosion). *Let $n>0$. Then $s_nI \to_{\beta_\lambda}^n r_n$. Moreover, $|s_nI| = O(n)$, $|r_n| = \Omega(2^n)$, s_nI is closed, and r_n is normal.* Proof p. 39

The evaluation of s_nI 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., Sacerdoti Coen, C.: On the Value of Variables. In: WoLLIC 2014. pp. 36–50 (2014)
10. Ariola, Z.M., Bohannon, A., Sabry, A.: Sequent calculi and abstract machines. ACM Trans. Program. Lang. Syst. 31(4) (2009)
11. Barendregt, H.P.: The Lambda Calculus – Its Syntax and Semantics, vol. 103. North-Holland (1984)
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)

Technical Appendix

A Rewriting Theory: Definitions, Notations, and Basic Results

Given a binary relation $\to_{\mathsf{r}}$ on a set I, the reflexive-transitive (resp. reflexive; transitive; reflexive-transitive and symmetric) closure of $\to_{\mathsf{r}}$ is denoted by $\to^*$ (resp. $\to_{\mathsf{r}}^{=}$; $\to_{\mathsf{r}}^{+}$; $\simeq_{\mathsf{r}}$). The transpose of $\to_{\mathsf{r}}$ is denoted by ${}_{\mathsf{r}}\!\leftarrow$. A ($\mathsf{r}$-)*derivation d from t to u*, denoted by $d\colon t \to_{\mathsf{r}}^* u$, is a finite sequence $(t_i)_{0 \leq i \leq n}$ of elements of I (with $n \in \mathbb{N}$) s.t. $t = t_0$, $u = t_n$ and $t_i \to_{\mathsf{r}} t_{i+1}$ for all $1 \leq i < n$;

The *number of* r*-steps of a derivation* d, *i.e.* its *length*, is denoted by $|d|_{\mathsf{r}} \coloneqq n$, or simply $|d|$. If $\to_{\mathsf{r}} = \to_1 \cup \to_2$ with $\to_1 \cap \to_2 = \emptyset$, $|d|_i$ is the number of $\to_i$-steps in d, for $i = 1, 2$. We say that:

- $t \in I$ is r*-normal* or a r*-normal form* if $t \not\to_{\mathsf{r}} u$ for all $u \in I$; $u \in I$ is a r*-normal form of* t if u is r-normal and $t \to_{\mathsf{r}}^* u$;
- $t \in I$ is r*-normalizable* if there is a r-normal $u \in I$ s.t. $t \to_{\mathsf{r}}^* u$; t is *strongly* r*-normalizable* if there is no infinite sequence $(t_i)_{i \in \mathbb{N}}$ s.t. $t_0 = t$ and $t_i \to_{\mathsf{r}} t_{i+1}$;
- a r-derivation $d\colon t \to_{\mathsf{r}}^* u$ is (r-)*normalizing* if u is r-normal;
- $\to_{\mathsf{r}}$ is *strongly normalizing* if all $t \in I$ is strongly r-normalizable;
- $\to_{\mathsf{r}}$ is *strongly confluent* if, for all $t, u, s \in I$ s.t. $s \;{}_{\mathsf{r}}\!\leftarrow t \to_{\mathsf{r}} u$ and $u \neq s$, there is $r \in I$ s.t. $s \to_{\mathsf{r}} r \;{}_{\mathsf{r}}\!\leftarrow u$; $\to_{\mathsf{r}}$ is *confluent* if $\to_{\mathsf{r}}^*$ is strongly confluent.

Let $\to_1, \to_2 \subseteq I \times I$. Composition of relations is denoted by juxtaposition: for instance, $t \to_1 \to_2 u$ means that there is $s \in I$ s.t. $t \to_1 s \to_2 u$; for any $n \in \mathbb{N}$, $t \to_1^n u$ means that there is a $\to_1$-derivation with length n ($t = u$ for $n = 0$). We say that $\to_1$ and $\to_2$ *strongly commute* if, for any $t, u, s \in I$ s.t. $u \;{}_1\!\leftarrow t \to_2 s$, one has $u \neq s$ and there is $r \in I$ s.t. $u \to_2 r \;{}_1\!\leftarrow s$. Note that if $\to_1$ and $\to_2$ strongly commute and $\to = \to_1 \cup \to_2$, then for any derivation $d\colon t \to^* u$ the sizes $|d|_1$ and $|d|_2$ are uniquely determined.

The following proposition collects some basic and well-known results of rewriting theory.

Proposition 23. *Let $\to_{\mathsf{r}}$ be a binary relation on a set I.*

1. *If $\to_{\mathsf{r}}$ is confluent then:*
   (a) *every* r*-normalizable term has a unique* r*-normal form;*
   (b) *for all $t, u \in I$, $t \simeq_{\mathsf{r}} u$ iff there is $s \in I$ s.t. $t \to_{\mathsf{r}}^* s \;{}_{\mathsf{r}}^{*}\!\leftarrow u$.*
2. *If $\to_{\mathsf{r}}$ is strongly confluent then $\to_{\mathsf{r}}$ is confluent and, for any $t \in I$, one has:*
   (a) *all normalizing* r*-derivations from t have the same length;*
   (b) *t is strongly* r*-normalizable if and only if t is* r*-normalizable.*

As all incarnations of Open CBV we consider are confluent, the use of Prop. 23.1 is left implicit.

For λ_{fire} and λ_{vsub}, we use Prop. 23.2 and the following more informative version of Hindley–Rosen Lemma, whose proof is just a more accurate reading of the proof in [11, Prop. 3.3.5.(i)]:

Lemma 24 (Strong Hindley–Rosen). *Let $\to = \to_1 \cup \to_2$ be a binary relation on a set I s.t. $\to_1$ and $\to_2$ are strongly confluent. If $\to_1$ and $\to_2$ strongly commute, then $\to$ is strongly confluent and, for any $t \in I$ and any normalizing derivations d and e from t, one has $|d| = |e|$, $|d|_1 = |e|_1$ and $|d|_2 = |e|_2$.*

B Omitted Proofs

B.1 Proofs of Section 2 (The Fireball Calculus)

In this section we start by recalling the relevant properties of the fireball calculus, that have been omitted from the paper for lack of space. For the sake of completeness we include all proofs, but most of them are actually taken from (the long versions of) our previous works [8,2].

Distinctive Properties of λ_{fire}. To prove the distinctive properties of λ_{fire} we need the following auxiliary lemma.

Lemma 25 (Values and inert terms are β_f-normal).

1. *Every abstraction is β_f-normal.*
2. *Every inert term is β_f-normal.*

Proof.

1. Immediate, since $\to_{\beta_f}$ does not reduce under λ's.
2. By induction on the definition of inert term i.
   - If $i = x$ then i is obviously β_f-normal.
   - If $i = i'\lambda x.t$ then i' and $\lambda x.t$ are β_f-normal by *i.h.* and Lemma 25.1 respectively, besides i' is not an abstraction, so i is β_f-normal.
   - Finally, if $i = i'i''$ then i' and i'' are β_f-normal by *i.h.*, moreover i' is not an abstraction, hence i is β_f-normal. □

The following proposition collects two main features of λ_{fire}, showing why it is interesting to study this calculus. Point 1 generalizes the property of Closed CbV, that we like to call *harmony*, for which a closed term is β_v-normal iff it is a value. Point 2 instead states that if the evaluation of a closed term t in the fireball calculus is *exactly* the evaluation of t in Closed CbV. This second point is an observation that never appeared in print before.

Proposition 1 (Distinctive Properties of λ_{fire}). *Let t be a term.* See p. 5

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.*

Proof.

1. **⇒:** Proof by induction on the term t. If t is a value then t is a fireball. Otherwise $t = us$ for some terms u and s. Since t is β_f-normal, then u and s are β_f-normal, and either u is not an abstraction or s is not a fireball. By induction hypothesis, u and s are fireballs. Summing up, u is either a variable or an inert term, and s is a fireball, therefore $t = us$ is an inert term and hence a fireball.

$\Leftarrow$: By hypothesis, t is either a value or an inert term. If t is a value, then it is β_f-normal by Lemma 25.1. Otherwise t is an inert term and then it is β_f-normal by Lemma 25.2.

2. **$\Rightarrow$:** By induction on the definition of $t \to_{\beta_f} u$. Cases:
   - *Step at the root*, *i.e.* $t = (\lambda x.s)f \mapsto_{\beta_f} s\{x{\leftarrow}f\} = u$. Since t is closed, then f is closed and hence cannot be an inert term, therefore f is a (closed) abstraction and thus $t = (\lambda x.s)f \mapsto_{\beta_\lambda} s\{x{\leftarrow}f\} = u$.
   - *Left Application*, *i.e.* $t = sr \to_{\beta_f} s'r = u$ with $s \to_{\beta_f} s'$. Since t is closed, s is closed and hence $s \to_{\beta_v} s'$ by *i.h.*, so $t = sr \to_{\beta_\lambda} s'r = u$.
   - *Right Application*, *i.e.* $t = rs \to_{\beta_f} rs' = u$ with $s \to_{\beta_f} s'$. Since t is closed, s is closed and hence $s \to_{\beta_v} s'$ by *i.h.*, so $t = rs \to_{\beta_\lambda} rs' = u$.

   $\Leftarrow$: We have $\to_{\beta_\lambda} \subseteq \to_{\beta_f}$ since variables and abstractions are fireballs. □

Operational Properties of λ_{fire}. The rewriting theory of the fireball calculus is very well behaved. The following propositions resumes its main properties.

See p. 6 **Proposition 2** (Operational Properties of λ_{fire}).
1. *$\to_{\beta_i}$ is strongly normalizing and strongly confluent.*
2. *$\to_{\beta_\lambda}$ and $\to_{\beta_i}$ strongly commute.*
3. *$\to_{\beta_f}$ is strongly confluent, and all β_f-normalizing derivations d from a term t (if any) 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.*

Proof. See [8, Proposition 3]. □

The Right-to-Left Strategy. Here we prove the the well-definedness of the right-to-left strategy.

See p. 6 **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.*

Proof.

1. ($\Rightarrow$) Immediate, as right contexts are in particular evaluation contexts, and thus $\to_{\mathtt{r}\beta_f} \subseteq \to_{\beta_f}$.
   ($\Leftarrow$) Let E the evaluation context of the rightmost redex of t. We show that E is a right context. By induction on E. Cases:
   (a) *Empty*, *i.e.* $E = \langle\cdot\rangle$. Then clearly E is a right context.
   (b) *Right Application*, *i.e.* $t = us$ and $E = uE'$. By *i.h.* E' is a right context in s and so is E with respect to t.
   (c) *Left Application*, *i.e.* $t = us$ and $E = E's$. By *i.h.* E' is a right context in u. Since E is the rightmost evaluation context, s is $\to_{\beta_f}$-normal, and so by open harmony (Prop. 1.1) it is a fireball. Therefore E is a right context.
2. By induction on t. Note that by completeness of $\to_{\mathtt{r}\beta_f}$ (Point 1) open harmony (Prop. 1.1) holds with respect to $\to_{\mathtt{r}\beta_f}$, *i.e.* a term is $\to_{\mathtt{r}\beta_f}$-normal iff it is a fireball. We use this fact implicitly in the following case analysis. Cases:

- *Value.* No redexes.
- *Application $t = us$.* By *i.h.*, there are two cases for s:
  (a) *s has exactly one $\rightarrow_{\mathtt{r}\beta_f}$ redex.* Then t has a $\rightarrow_{\mathtt{r}\beta_f}$ redex, because $u\langle\cdot\rangle$ is an evaluation context. Moreover, no $\rightarrow_{\mathtt{r}\beta_f}$ redex for t can lie in u, because by open harmony s is not a fireball, and so $\langle\cdot\rangle s$ is not a right context.
  (b) *s has no $\rightarrow_{\mathtt{r}\beta_f}$ redexes.* Then s is a fireball. Consider u. By *i.h.*, there are two cases:
    i. *u has exactly one $\rightarrow_{\mathtt{r}\beta_f}$ redex.* Then t has a $\rightarrow_{\mathtt{r}\beta_f}$ redex, because $\langle\cdot\rangle s$ is an evaluation context and s is a fireball. Uniqueness follows from the fact that s has no $\rightarrow_{\mathtt{r}\beta_f}$ redexes.
    ii. *u has no $\rightarrow_{\mathtt{r}\beta_f}$ redexes.* By open harmony u is a fireball, and there are two cases:
      - *u is an inert term i or a variable x.* Then t is a fireball.
      - *u is an abstraction $\lambda x.r$.* Then $t = (\lambda x.r)s$ is a $\rightarrow_{\mathtt{r}\beta_f}$-redex, because s is a fireball. Moreover, there are no other $\rightarrow_{\mathtt{r}\beta_f}$ redexes, because evaluation does not go under abstractions and s is a fireball. □

Open Size Explosion. The proof of open size explosion is a particularly simple induction on the index of the size exploding family.

Proposition 5 (Open Size Explosion). *Let $n \in \mathbb{N}$. Then $t_n \rightarrow_{\beta_i}^n i_n$, moreover $|t_n| = O(n)$, $|i_n| = \Omega(2^n)$, and i_n is an inert term.* See p. 6

Proof. By induction on n. The base case is immediate. The inductive case: $t_{n+1} = (\lambda x.xx)t_n \rightarrow_{\beta_i}^n (\lambda x.xx)i_n \rightarrow_{\beta_i} i_n i_n = i_{n+1}$, where the first sequence is obtained by the *i.h.* The bounds on the sizes are immediate, as well as the fact that i_{n+1} is inert. □

Circumventing Open Size Explosion. To prove that the substitution of inert terms can be avoided we need two auxiliary simple but technical lemmas about substitution, fireballs, and reductions.

Lemma 26 (Fireballs are Closed Under Substitution and Anti-Substitution of Inert Terms). *Let t be a term and i be an inert term.*
1. *$t\{x\leftarrow i\}$ is an abstraction iff t is an abstraction.*
2. *$t\{x\leftarrow i\}$ is an inert term iff t is an inert term;*
3. *$t\{x\leftarrow i\}$ is a fireball iff t is a fireball.*

Proof.

1. If $t\{x\leftarrow i\} = \lambda y.s$ then we can suppose without loss of generality that $y \notin \mathtt{fv}(i) \cup \{x\}$ and thus there is r such that $s = r\{x\leftarrow i\}$, hence $t\{x\leftarrow i\} = \lambda y.(r\{x\leftarrow i\}) = (\lambda y.r)\{x\leftarrow i\}$, therefore $t = \lambda y.r$ is an abstraction.
   Conversely, if $t = \lambda y.s$ then we can suppose without loss of generality that $y \notin \mathtt{fv}(i) \cup \{x\}$ and thus $t\{x\leftarrow i\} = \lambda y.(s\{x\leftarrow i\})$ which is an abstraction.

2. ($\Rightarrow$): By induction on the inert structure of $t\{x{\leftarrow}i\}$. Cases:
   - *Variable*, *i.e.* $t\{x{\leftarrow}i\} = y$, possibly with $x = y$. Then $t = x$ or $t = y$, and in both cases t is inert.
   - *Compound Inert*, *i.e.* $t\{x{\leftarrow}i\} = i'f$. If t is a variable then it is inert. Otherwise it is an application $t = us$, and so $u\{x{\leftarrow}i\} = i'$ and $s\{x{\leftarrow}i\} = f$. By *i.h.*, u is an inert term. Consider f. Two cases:
     (a) f is an abstraction. Then by Point 1 s is an abstraction.
     (b) f is an inert term. Then by *i.h.* s is an inert term.
     In both cases s is a fireball, and so $t = us$ is an inert term.

   ($\Leftarrow$): By induction on the inert structure of t. Cases:
   - *Variable*, *i.e.* either $t = x$ or $t = y$: in the first case $t\{x{\leftarrow}i\} = i$, in the second case $t\{x{\leftarrow}i\} = y$; in both cases $t\{x{\leftarrow}i\}$ is an inert term.
   - *Compound Inert*, *i.e.* $t = i'f$. Then $t\{x{\leftarrow}i\} = i'\{x{\leftarrow}i\}f\{x{\leftarrow}i\}$. By *i.h.*, $i'\{x{\leftarrow}i\}$ is an inert term. Concerning f, there are two cases:
     (a) f is an abstraction. Then by Point 1 $f\{x{\leftarrow}i\}$ is an abstraction.
     (b) f is an inert term. Then by *i.h.* $f\{x{\leftarrow}i\}$ is an inert term.
     In both cases $f\{x{\leftarrow}i\}$ is a fireball, and hence $t\{x{\leftarrow}i\} = i'\{x{\leftarrow}i\}f\{x{\leftarrow}i\}$ is an inert term.
3. Immediate consequence of Lemmas 26.1-2, since every fireball is either an abstraction or an inert term. □

Lemma 27 (Substitution of Inert Terms Does Not Create β_f-Redexes). *Let t, u be terms and i be an inert term. There is a term s such that:*
1. *if $t\{x{\leftarrow}i\} \to_{\beta_\lambda} u$ then $t \to_{\beta_\lambda} s$ and $s\{x{\leftarrow}i\} = u$;*
2. *if $t\{x{\leftarrow}i\} \to_{\beta_i} u$ then $t \to_{\beta_i} s$ and $s\{x{\leftarrow}i\} = u$.*

Proof. We prove the two points by induction on the evaluation context closing the root redex. Cases:

- *Step at the root*:
  1. *Abstraction Step*, *i.e.* $t\{x{\leftarrow}i\} \coloneqq (\lambda y.r\{x{\leftarrow}i\})q\{x{\leftarrow}i\} \mapsto_{\beta_\lambda} r\{x{\leftarrow}i\}\{y{\leftarrow}q\{x{\leftarrow}i\}\} \eqqcolon u$. By Lemma 26.1, q is an abstraction, since $q\{x{\leftarrow}i\}$ is an abstraction by hypothesis. Then $t = (\lambda y.r)q \mapsto_{\beta_\lambda} r\{y{\leftarrow}q\}$. Then $s \coloneqq r\{x{\leftarrow}q\}$ verifies the statement, as $s\{x{\leftarrow}i\} = (r\{y{\leftarrow}q\})\{x{\leftarrow}i\} = r\{x{\leftarrow}i\}\{y{\leftarrow}q\{x{\leftarrow}i\}\} = u$.
  2. *Inert Step*, identical to the abstraction subcase, just replace *abstraction* with *inert term* and the use of Lemma 26.1 with the use of Lemma 26.2.
- *Application Left*, *i.e.* $t = rq$ and reduction takes place in r:
  1. *Abstraction Step*, *i.e.* $t\{x{\leftarrow}i\} \coloneqq r\{x{\leftarrow}i\}q\{x{\leftarrow}i\} \to_{\beta_\lambda} pq\{x{\leftarrow}i\} \eqqcolon u$. By *i.h.* there is a term s' such that $p = s'\{x{\leftarrow}i\}$ and $r \to_{\beta_\lambda} s'$. Then $s \coloneqq s'q$ satisfies the statement, as $s\{x{\leftarrow}i\} = (s'q)\{x{\leftarrow}i\} = s'\{x{\leftarrow}i\}q\{x{\leftarrow}i\} = u$.
  2. *Inert Step*, identical to the abstraction subcase.
- *Application Right*, *i.e.* $t = rq$ and reduction takes place in q. Identical to the *application left* case, just switch left and right. □

See p. 6

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 $t\{x{\leftarrow}i\} \to_{\beta_f} u\{x{\leftarrow}i\}$.*

Proof. The right-to-left direction is Lemma 27, since $\to_{\beta_f} = \to_{\beta_\lambda} \cup \to_{\beta_i}$. The left-to-right direction is proved by induction on the definition of $t \to_{\beta_f} u$. Cases:

– *Step at the root*:
  1. *Abstraction Step, i.e.* $t = (\lambda y.s)r \mapsto_{\beta_\lambda} s\{y{\leftarrow}r\} = u$ where r is an abstraction. We can suppose without loss of generality that $y \in \mathtt{fv}(i) \cup \{x\}$. Note that $r\{x{\leftarrow}i\}$ is an abstraction, according to Lemma 26.1. Then, $t\{x{\leftarrow}i\} = (\lambda y.s\{x{\leftarrow}i\})r\{x{\leftarrow}i\} \mapsto_{\beta_\lambda} s\{x{\leftarrow}i\}\{y{\leftarrow}r\{x{\leftarrow}i\}\} = s\{y{\leftarrow}r\}\{x{\leftarrow}i\} = u\{x{\leftarrow}i\}$.
  2. *Inert Step, i.e.* $t = (\lambda y.s)i' \mapsto_{\beta_i} s\{y{\leftarrow}i'\} = u$ where i' is an inert term. We can suppose without loss of generality that $y \in \mathtt{fv}(i) \cup \{x\}$. According to Lemma 26.2, $r\{x{\leftarrow}i\}$ is inert. Then, $t\{x{\leftarrow}i\} = (\lambda y.s\{x{\leftarrow}i\})i'\{x{\leftarrow}i\} \mapsto_{\beta_i} s\{x{\leftarrow}i\}\{y{\leftarrow}i'\{x{\leftarrow}i\}\} = s\{y{\leftarrow}i'\}\{x{\leftarrow}i\} = u\{x{\leftarrow}i\}$.

– *Application Left, i.e.* $t = sr \to_{\beta_f} s'r = u$ with $s \to_{\beta_f} s'$. By *i.h.*, $s\{x{\leftarrow}i\} \to_{\beta_f} s'\{x{\leftarrow}i\}$, hence $t\{x{\leftarrow}i\} = s\{x{\leftarrow}i\}r\{x{\leftarrow}i\} \to_{\beta_f} s'\{x{\leftarrow}i\}r\{x{\leftarrow}i\} = u\{x{\leftarrow}i\}$.

– *Application Right, i.e.* $t = rs \to_{\beta_f} rs' = u$ with $s \to_{\beta_f} s'$. By *i.h.*, $s\{x{\leftarrow}i\} \to_{\beta_f} s'\{x{\leftarrow}i\}$, hence $t\{x{\leftarrow}i\} = r\{x{\leftarrow}i\}s\{x{\leftarrow}i\} \to_{\beta_f} r\{x{\leftarrow}i\}s'\{x{\leftarrow}i\} = u\{x{\leftarrow}i\}$. □

B.2 Proofs of Section 3 (Preliminaries on Abstract Machines, Implementations, and Complexity Analysis)

Here we provide the abstract proof of Thm. 9, stating that the conditions required to an implementation system (Def. 8) indeed imply that the machine implements the strategy via the decoding (in the sense of Def. 7).

The *executions-to-derivations* part of the implementation theorem is easy to prove, essentially *β-projection* and *overhead transparency* allow to project a single transition, and the projection of executions is obtained as a simple induction.

The *derivations-to-executions* part is a bit more delicate, instead, because the simulation of β-steps has to be done *up to* overhead transitions. The following lemma shows how the conditions for implementation systems allow to do that. Interestingly all five conditions of Def. 8 are used in the proof.

Lemma 28 (One-Step Simulation). *Let* $\mathtt{M}$, $\to$, *and* $\underline{\cdot}$ *be a machine, a strategy, and a decoding forming an implementation system. For any state* s *of* $\mathtt{M}$, *if* $\underline{s} \to u$ *then there is a state* s' *of* $\mathtt{M}$ *such that* $s \rightsquigarrow_{\mathtt{o}}^* \rightsquigarrow_{\beta} s'$.

Proof. According to Def. 8, since $(\mathtt{M}, \to, \underline{\cdot})$ is an implementation system, 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.

For any state s of $\mathtt{M}$, let $\mathtt{nf}_{\mathtt{o}}(s)$ be the state that is the normal form of s with respect to $\rightsquigarrow_{\mathtt{o}}$: such a state exists and is unique because overhead transitions terminate (Point 3) and $\mathtt{M}$ is deterministic (Point 4). Since $\rightsquigarrow_{\mathtt{o}}$ is mapped on identities (Point 2), one has $\underline{\mathtt{nf}_{\mathtt{o}}(s)} = \underline{s}$. As $\underline{s}$ is not $\to$-normal by hypothesis, the progress property (Point 5) entails that $\mathtt{nf}_{\mathtt{o}}(s)$ is not final, therefore $s \rightsquigarrow_{\mathtt{o}}^{*} \mathtt{nf}_{\mathtt{o}}(s) \rightsquigarrow_{\beta} s'$ for some state s', and thus $\underline{s} = \underline{\mathtt{nf}_{\mathtt{o}}(s)} \to \underline{s'}$ by β-projection (Point 1). According to the determinism of $\to$ (Point 4), one obtains $\underline{s'} = u$. □

Now, the one-step simulation can be extended to the a simulation of derivations by an easy induction on the length of the derivation.

See p. 8 **Theorem 9** (Sufficient Condition for Implementations)**.** *Let* $(\mathtt{M}, \to, \underline{\cdot})$ *be an* implementation system*. Then,* $\mathtt{M}$ *implements* $\to$ *via* $\underline{\cdot}$*.*

Proof. According to Def. 7, given a λ-term t, we have to show that:

(i) *Executions to Derivations with β-matching*: for any $\mathtt{M}$-execution $\rho : t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s$ there exists a $\to$-derivation $d : t \to^{*} \underline{s}$ such that $|d| = |\rho|_{\beta}$.
(ii) *Derivations to Executions with β-matching*: for every $\to$-derivation $d : t \to^{*} u$ there exists a $\mathtt{M}$-execution $\rho : t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s$ such that $\underline{s} = u$ and $|d| = |\rho|_{\beta}$.

Proof of Point (i)*:* by induction on $|\rho|_{\beta} \in \mathbb{N}$.

If $|\rho|_{\beta} = 0$ then $\rho \colon t^{\circ} \rightsquigarrow_{\mathtt{o}}^{*} s$ and hence $\underline{t^{\circ}} = \underline{s}$ by overhead transparency (Point 2 of Def. 8). Moreover, $t = \underline{t^{\circ}}$ since decoding is the inverse of compilation on initial states, therefore we are done by taking the empty (*i.e.* without steps) derivation d with starting (and end) term t.

Suppose $|\rho|_{\beta} > 0$: then, $\rho \colon t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s$ is the concatenation of an execution $\rho' \colon t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s'$ followed by an execution $\rho'' \colon s' \rightsquigarrow_{\beta} s'' \rightsquigarrow_{\mathtt{o}}^{*} s$. By *i.h.* applied to ρ', there exists a derivation $d' \colon t \to^{*} \underline{s'}$ with $|\rho'|_{\beta} = |d'|$. By β-projection (Point 1 of Def. 8) and overhead transparency (Point 2 of Def. 8) applied to ρ'', one has $d'' \colon \underline{s'} \to \underline{s''} = \underline{s}$. Therefore, the derivation d defined as the concatenation of d' and d'' is such that $d \colon t \to^{*} \underline{s}$ and $|d| = |d'| + |d''| = |\rho'|_{\beta} + 1 = |\rho|_{\beta}$.

Proof of Point (ii)*:* by induction on $|d| \in \mathbb{N}$.

If $|d| = 0$ then $t = u$. Since decoding is the inverse of compilation on initial states, one has $\underline{t^{\circ}} = t$. We are done by taking the empty (*i.e.* without transitions) execution ρ with initial (and final) state t°.

Suppose $|d| > 0$: so, $d \colon t \to^{*} u$ is the concatenation of a derivation $d' \colon t \to^{*} u'$ followed by the step $u' \to u$. By *i.h.*, there exists a $\mathtt{M}$-execution $\rho' \colon t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s'$ such that $\underline{s'} = u'$ and $|d'| = |\rho'|_{\beta}$. According to the one-step simulation (Lemma 28, since $\underline{s'} \to u$ and $(\mathtt{M}, \to, \underline{\cdot})$ is an implementation system), there is a state s of $\mathtt{M}$ such that $s' \rightsquigarrow_{\mathtt{o}}^{*} \rightsquigarrow_{\beta} s$ and $\underline{s} = u$. Therefore, the execution $\rho \colon t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s' \rightsquigarrow_{\mathtt{o}}^{*} \rightsquigarrow_{\beta} s$ is such that $|\rho|_{\beta} = |\rho'|_{\beta} + 1 = |d'| + 1 = |d|$. □

B.3 Proofs of Section 4 (Easy GLAMOUr)

First we prove the invariants of the Easy GLAMOUr, and then we use them to prove that it forms an implementation system with respect to right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ in the fireball calculus (via the decoding).

See p. 12 **Lemma 11** (Easy GLAMOUr 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, $\overline{t}$, π, or E then x may occur only in $\overline{u}$;*

3. 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;*

4. Contextual Decoding*: $C_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context.*

Proof. By induction on the length of the execution leading to the reachable state. In an initial state all the invariants trivially hold. For a non-empty execution the proof for every invariant is by case analysis on the last transition, using the *i.h.*.

1. *Name.* Cases:
   (a) $s' = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{c_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. Both points follow immediately from the *i.h.*
   (b) $s' = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{c_2} (D, \overline{t}, \lambda x.\overline{u}@\epsilon : \pi, E) = s$. Both points follow immediately from the *i.h.*
   (c) $s' = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{c_3} (D, \overline{t}, x@\pi' : \pi, E) = s$ with $E(x) = \bot$ or $E(x) = y@\pi''$. Both points follow immediately from the *i.h.*
   (d) $s' = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta} (D, \overline{t}, \pi, [x{\leftarrow}\phi]E) = s$. Point 1 for the new entry in the environment follows from the *i.h.* for Point 2, for the other entries from the *i.h.* for Point 1. Point 2 follows from its *i.h.*
   (e)
$$\begin{aligned} s' &= (D, x, \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2) \\ &\rightsquigarrow_{s} (D, (\lambda y.\overline{u})^{\alpha}, \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2) = s. \end{aligned}$$
   Point 1 follows from its *i.h.*. Point 2 for the new code is guaranteed by the α-renaming operation $(\lambda y.\overline{u})^{\alpha}$, the rest follows from its *i.h.*
2. *Fireball Item.* Cases:
   (a) $s' = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{c_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. It follows from the *i.h.*
   (b) $s' = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{c_2} (D, \overline{t}, \lambda x.\overline{u}@\epsilon : \pi, E) = s$. For $\lambda x.\overline{u}@\epsilon$ we have that $\underline{\lambda x.\overline{u}@\epsilon}$ and $\underline{\lambda x.\overline{u}@\epsilon}\downarrow_E = (\lambda x.\overline{u})\downarrow_E = \lambda x.\overline{u}\downarrow_E$ are abstractions, and hence fireballs. For all other items the invariant follows from the *i.h.*
   (c) $s' = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{c_3} (D, \overline{t}, x@\pi' : \pi, E) = s$ with $E(x) = \bot$ or $E(x) = y@\pi''$. For $x@\pi'$, we have that $\underline{x@\pi'} = \langle x\rangle\underline{\pi'}$ and $\underline{x@\pi'}\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi'}\downarrow_E)$. By *i.h.*, $\underline{\phi'}$ is a fireball for every item ϕ' in π'. Therefore, $\underline{x@\pi'}$ is an inert term. Concerning $\underline{x@\pi'}\downarrow_E$, there are two subcases:
      i. $E(x) = y@\pi''$ *i.e.* $E := E_1[x{\leftarrow}y@\pi'']E_2$. By Lemma 11.1.1, every ES in E binds a different variable, so $x\downarrow_E = x\downarrow_{E_1[x\leftarrow y@\pi'']E_2} = x\downarrow_{E_1}\{x{\leftarrow}\underline{y@\pi''}\}\downarrow_{E_2} = \underline{y@\pi''}\downarrow_{E_2} = \underline{y@\pi''}\downarrow_E$, that by *i.h.* is an inert term. Moreover, the *i.h.* also gives that $\underline{\phi'}\downarrow_E$ is a fireball for every item ϕ' in π'. Therefore $\underline{x@\pi'}\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi'}\downarrow_E)$ is an inert term.
      ii. $E(x) = \bot$. Similar to the previous case. By hypothesis, we have $x\downarrow_E = x$. As before, by *i.h.* $\underline{\phi'}\downarrow_E$ is a fireball for every item ϕ' in π'. So, $\underline{x@\pi'}\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi'}\downarrow_E) = \langle x\rangle(\underline{\pi'}\downarrow_E)$ is an inert term.

For all other items in s the invariant follows from the *i.h.*

(d) $s' = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta} (D, \overline{t}, \pi, [x{\leftarrow}\phi]E) = s$. By Lemma 11.1.2 x may occur only in $\overline{t}$. Thus the substitution $\downarrow_{[x\leftarrow\phi]E}$ acts exactly as $\downarrow_E$ on every item in s. Then the invariant follows from the *i.h.*

(e)

$$\begin{aligned} s' &= (D, x, \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2) \\ &\rightsquigarrow_{\mathtt{s}} (D, (\lambda y.\overline{u})^{\alpha}, \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2) = s. \end{aligned}$$

It follows from the *i.h.*

3. *Contextual Decoding.* Cases:

(a) $s' = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{\mathtt{c}_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. By *i.h.* $C_{s'} = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context, as well as $\overline{u}\downarrow_E\langle\cdot\rangle$. Then their composition $(\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\overline{u}\downarrow_E\langle\cdot\rangle\rangle$ $= \underline{D}\langle\langle\overline{u}\langle\cdot\rangle\rangle\underline{\pi}\rangle\downarrow_E = C_s$ is a right context.

(b) $s' = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{\mathtt{c}_2} (D, \overline{t}, \lambda x.\overline{u}@\epsilon : \pi, E) = s$. By *i.h.* $C_{s'} = \underline{D : \overline{t}\Diamond\pi}\downarrow_E = \underline{D}\langle\langle\overline{t}\langle\cdot\rangle\rangle\underline{\pi}\rangle\downarrow_E$ is a right context, that implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well. So, $C_{s'} = \underline{D}\langle\underline{\lambda x.\overline{u}@\epsilon : \pi}\rangle\downarrow_E = \underline{D}\langle\langle\langle\cdot\rangle\lambda x.\overline{u}\rangle\underline{\pi}\rangle\downarrow_E$ $= (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\langle\cdot\rangle\lambda x.\overline{u}\downarrow_E\rangle$ is a right context, because it is the composition of right context, given that $\lambda x.\overline{u}\downarrow_E$ is a fireball.

(c) $s' = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{\mathtt{c}_3} (D, \overline{t}, x@\pi' : \pi, E) = s$ with $E(x) = \bot$ or with $E(x) = y@\pi''$. By *i.h.* $C_{s'} = \underline{D : \overline{t}\Diamond\pi}\langle\underline{\pi'}\rangle\downarrow_E = \underline{D}\langle\langle\overline{t}(\underline{\pi'})\rangle\underline{\pi}\rangle\downarrow_E$ is a right context, that implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well. Then $C_s = \underline{D}\langle\underline{x@\pi' : \pi}\rangle\downarrow_E = \underline{D}\langle\langle\langle\cdot\rangle\underline{x@\pi'}\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\langle\cdot\rangle\underline{x@\pi'}\downarrow_E\rangle$ is a right context, because it is the composition of right context, given that $\underline{x@\pi'}\downarrow_E$ is a fireball by Lemma 11.3.

(d) $s' = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta} (D, \overline{t}, \pi, [x{\leftarrow}\phi]E) = s$. By *i.h.* $C_{s'} = \underline{D}\langle\underline{\phi : \pi}\rangle\downarrow_E$ is a right context, that implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well. Now, note that $C_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_{[x\leftarrow\phi]E} = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ because by Lemma 11.1.2 x may occur only in $\overline{t}$, and so the substitution $\downarrow_{[x\leftarrow\phi]E}$ acts on every code in D and π exactly as $\downarrow_E$.

(e)

$$\begin{aligned} s' &= (D, x, \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2) \\ &\rightsquigarrow_{\mathtt{s}} (D, (\lambda y.\overline{u})^{\alpha}, \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2) = s. \end{aligned}$$

It follows by the *i.h.* because $C_{s'} = C_s$, as the only component that changes is the code. □

Note 29. Given a machine $\mathtt{M}$, a transition is a binary relation on the set of states of $\mathtt{M}$. Given two transitions $\rightsquigarrow_{\mathtt{r}_1}$ and $\rightsquigarrow_{\mathtt{r}_2}$, we set $\rightsquigarrow_{\mathtt{r}_1,\mathtt{r}_2} := \rightsquigarrow_{\mathtt{r}_1} \cup \rightsquigarrow_{\mathtt{r}_2}$ (also denoted by $\rightsquigarrow_{\mathtt{r}_{1,2}}$ or simply $\rightsquigarrow_{\mathtt{r}}$).

Conditions for an Implementation System. We now prove that the Easy GLAMOUr satisfies the conditions for an implementation system with respect to $\rightarrow_{\mathtt{r}\beta_f}$. First, we deal with the two conditions about the projection of transitions on the calculus.

Lemma 30 (Easy GLAMOUr β-Projection and Overhead Transparency). *Let s be a reachable state.*

1. Overhead Transparency*: if $s \rightsquigarrow_{\texttt{s},\texttt{c}_{1,2,3}} s'$ (see Note 29 for the meaning of $\rightsquigarrow_{\texttt{s},\texttt{c}_{1,2,3}}$) then $\underline{s} = \underline{s'}$;*
2. β-Projection*: if $s \rightsquigarrow_{\beta} s'$ then $\underline{s} \rightarrow_{\mathtt{r}\beta_f} \underline{s'}$.*

Proof. Transitions:

1. $s = (D, \bar{t}\overline{u}, \pi, E) \rightsquigarrow_{\texttt{c}_1} (D : \bar{t}\lozenge\pi, \overline{u}, \epsilon, E) = s'$. Then

$$\begin{aligned}\underline{s} &= \underline{D}\langle\langle \bar{t}\overline{u}\rangle\underline{\pi}\rangle\downarrow_E\\ &= \underline{D : \bar{t}\lozenge\pi}\langle\overline{u}\rangle\downarrow_E\\ &= \underline{D : \bar{t}\lozenge\pi}\langle\langle\overline{u}\rangle\underline{\epsilon}\rangle\downarrow_E = \underline{s'}\end{aligned}$$

2. $s = (D : \bar{t}\lozenge\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{\texttt{c}_2} (D, \bar{t}, \lambda x.\overline{u}@\epsilon : \pi, E) = s'$. Then

$$\begin{aligned}\underline{s} &= \underline{D : \bar{t}\lozenge\pi}\langle\langle\lambda x.\overline{u}\rangle\underline{\epsilon}\rangle\downarrow_E\\ &= \underline{D}\langle\langle\bar{t}(\langle\lambda x.\overline{u}\rangle\underline{\epsilon})\rangle\underline{\pi}\rangle\downarrow_E\\ &= \underline{D}\langle\langle\bar{t}\rangle\underline{\lambda x.\overline{u}@\epsilon : \pi}\rangle\downarrow_E = \underline{s'}\end{aligned}$$

3. $s = (D : \bar{t}\lozenge\pi, x, \pi', E) \rightsquigarrow_{\texttt{c}_3} (D, \bar{t}, x@\pi' : \pi, E) = s'$ with $E(x) = \bot$ or $E(x) = y@\pi''$. Then

$$\begin{aligned}\underline{s} &= \underline{D : \bar{t}\lozenge\pi}\langle\langle x\rangle\underline{\pi'}\rangle\downarrow_E\\ &= \underline{D}\langle\langle\bar{t}(\langle x\rangle\underline{\pi'})\rangle\underline{\pi}\rangle\downarrow_E\\ &= \underline{D}\langle\langle\bar{t}\rangle\underline{x@\pi' : \pi}\rangle\downarrow_E = \underline{s'}\end{aligned}$$

4. $s = (D, \lambda x.\bar{t}, \phi : \pi, E) \rightsquigarrow_{\beta} (D, \bar{t}, \pi, [x{\leftarrow}\phi]E) = s'$. Then

$$\begin{aligned}\underline{s} &= \underline{D}\langle\langle\lambda x.\bar{t}\rangle\underline{\phi : \pi}\rangle\downarrow_E\\ &= \underline{D}\langle\langle(\lambda x.\bar{t})\underline{\phi}\rangle\underline{\pi}\rangle\downarrow_E\\ &\rightarrow_{\mathtt{r}\beta_f} \underline{D}\langle\langle\bar{t}\{x{\leftarrow}\underline{\phi}\}\rangle\underline{\pi}\rangle\downarrow_E\\ &= \underline{D}\langle\langle\bar{t}\rangle\underline{\pi}\rangle\{x{\leftarrow}\underline{\phi}\}\downarrow_E\\ &= \underline{D}\langle\langle\bar{t}\rangle\underline{\pi}\rangle\downarrow_{[x\leftarrow\phi]E} \quad = \underline{s'}\end{aligned}$$

   where the rewriting step takes place because
   (a) $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context by Lemma 11.4;
   (b) $\underline{\phi}$ is a fireball by Lemma 11.3.
   Moreover, the meta-level substitution $\{x{\leftarrow}\underline{\phi}\}$ can be extruded (in the equality step after the rewriting) without renaming x, because by Lemma 11.1.2 x does not occur in D nor π.

5.
$$\begin{aligned}s &= (D, x, \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2)\\ &\rightsquigarrow_{\texttt{s}} (D, (\lambda y.\overline{u})^{\alpha}, \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2) = s'.\end{aligned}$$

   Let $E' \coloneqq E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2$. Then

$$\begin{aligned}\underline{s} &= \underline{D}\langle\langle x\rangle\underline{\pi}\rangle\downarrow_{E'}\\ &= \underline{D\downarrow_{E'}}\langle\langle x\downarrow_{E'}\rangle\underline{\pi\downarrow_{E'}}\rangle\\ &= \underline{D\downarrow_{E'}}\langle\langle\lambda y.\overline{u}\downarrow_{E'}\rangle\underline{\pi\downarrow_{E'}}\rangle\\ &= \underline{D}\langle\langle\lambda y.\overline{u}\rangle\underline{\pi}\rangle\downarrow_{E'} \qquad = \underline{s'}\end{aligned}$$

□

We also need a lemma for the progress condition.

Lemma 31 (Easy GLAMOUr Progress). *Let s be a reachable final state. Then $\underline{s}$ is fireball, i.e. it is β_f-normal.*

Proof. An immediate inspection of the transitions shows that in a final state the code cannot be an application and the dump is necessarily empty. In fact, final states have one of the following two shapes:

1. *Top-Level Unapplied Abstraction, i.e.* $s = (\epsilon, \lambda x.\bar{t}, \epsilon, E)$. Then $\underline{s} = (\lambda x.\bar{t})\downarrow_E = \lambda x.\bar{t}\downarrow_E$ that is a fireball.
2. *Top-Level Free Variable or Inert Term with Free Head, i.e.* $s = (\epsilon, x, \pi, E)$ with $E(x) = \perp$. Then $\underline{s} = (\langle x\rangle\underline{\pi})\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi}\downarrow_E) = \langle x\rangle(\underline{\pi}\downarrow_E)$. Now, by the fireball item invariant (Lemma 11.3) every element of $\underline{\pi}\downarrow_E$ is a fireball, and so $\langle x\rangle(\underline{\pi}\downarrow_E)$ is an inert term, *i.e.* a fireball. □

Finally, we obtain the implementation theorem.

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

Proof. According to Thm. 9, it is enough to show that the Easy GLAMOUr and the right-to-left evaluation $\rightarrow_{\mathtt{r}\beta_f}$ and the decoding $\underline{\cdot}$ form an implementation system, *i.e.* that the five conditions in Def. 8 hold. Note that substitution ($\rightsquigarrow_{\mathtt{s}}$) and commutative ($\rightsquigarrow_{\mathtt{c}_{1,2,3}}$) transitions are considered as overhead transitions.

1. *β-Projection*: $s \rightsquigarrow_\beta s'$ implies $\underline{s} \rightarrow \underline{s'}$ by Lemma 30.2.
2. *Overhead Transparency*: $s \rightsquigarrow_{\mathtt{s},\mathtt{c}_{1,2,3}} s'$ implies $\underline{s} = \underline{s'}$ by Lemma 30.1 (recall that $\rightsquigarrow_{\mathtt{s},\mathtt{c}_{1,2,3}} = \rightsquigarrow_{\mathtt{s}} \cup \rightsquigarrow_{\mathtt{c}_1} \cup \rightsquigarrow_{\mathtt{c}_2} \cup \rightsquigarrow_{\mathtt{c}_3}$ according to Note 29).
3. *Overhead Transitions Terminate*: Termination of $\rightsquigarrow_{\mathtt{s},\mathtt{c}_{1,2,3}}$ is given by forthcoming Lemma 16 and Cor. 15, which are postponed because they actually give precise complexity bounds, not just termination.
4. *Determinism*: The Easy GLAMOUr machine is deterministic, as it can be seen by an easy inspection of the transitions (see Fig. 2). Lemma 3.2 proves that $\rightarrow_{\mathtt{r}\beta_f}$ is deterministic.
5. *Progress*: Let s be an Easy GLAMOUr final state. By Lemma 31, $\underline{s}$ is a β_f-normal term, in particular it is $\rightarrow_{\mathtt{r}\beta_f}$-normal because $\rightarrow_{\mathtt{r}\beta_f} \subseteq \rightarrow_{\beta_f}$. □

B.4 Proofs of Section 5 (Complexity Analysis of the Easy GLAMOUr)

See p. 12 **Lemma 13** (Subterm Invariant). *Let $\rho : t_0^\circ \rightsquigarrow^* (D, \bar{t}, \pi, E)$ be a Easy GLAMOUr execution. If $\lambda x.\overline{u}$ is a subterm of D, $\bar{t}$, π, or E then it is a subterm of t_0.*

Proof. First of all, let us be precise about *subterms*: for us, $\overline{u}$ is a subterm of t_0 if it does so up to variable names, both free and bound (and so the distinction between terms and codes is irrelevant). More precisely: define t^- as t in which all variables (including those appearing in binders) are replaced by a fixed symbol $*$. Then, we will consider u to be a subterm of t whenever u^- is a subterm of t^- in the usual sense. The key property ensured by this definition is that the size $|\overline{u}|$ of $\overline{u}$ is bounded by $|\overline{t}|$.

Now, the proof is by induction on the length of the execution leading to the reachable state. In an initial state the invariant trivially holds. For a non-empty execution the proof is by a straightforward case analysis on the last transition, always relying on the *i.h.* □

Lemma 14 (Free Occurrences Invariant). *Let $\rho : t_0^\circ \rightsquigarrow^* s$ be a Easy GLAMOUr execution. Then $|s|_{\mathsf{free}} \leq |t_0|_{\mathsf{free}} + |t_0| \cdot |\rho|_\beta - |\rho|_{\mathsf{s}}$.* See p. 13

Proof. By induction on $|\rho|$. Case $|\rho| = 0$ is obvious, since $t_0^\circ = s$. Otherwise $\sigma : t_0^\circ \rightsquigarrow^* s'$ and ρ extends σ with $s' \rightsquigarrow s$. By *i.h.*, $|s'|_{\mathsf{free}} \leq |t_0|_{\mathsf{free}} + |t_0| \cdot |\sigma|_\beta - |\sigma|_{\mathsf{s}}$. Cases (the notation refers to the transitions of the machine, in Fig. 2):

– *the last transition is a substitution transition.* We have to show $|s|_{\mathsf{free}} \leq |\overline{t}|_{\mathsf{free}} + |t_0| \cdot |\rho|_\beta - |\rho|_{\mathsf{s}}$. It follows from the *i.h.* and
  - $|s|_{\mathsf{free}} = |s'|_{\mathsf{free}} - 1$ because dump and stack do not change and the code changes from a variable (of measure 1) to an abstraction (of measure 0);
  - $|\rho|_\beta = |\sigma|_\beta$;
  - $|\rho|_{\mathsf{s}} = |\sigma|_{\mathsf{s}} + 1$;

– *the last transition is a β-transition.* For $\rightsquigarrow_\beta$:

$$\begin{aligned}
|\rho|_{\mathsf{free}} &= |D|_{\mathsf{free}} + |\pi|_{\mathsf{free}} + |\overline{t}|_{\mathsf{free}} \\
&\leq |D|_{\mathsf{free}} + |f : \pi|_{\mathsf{free}} + |\overline{t}|_{\mathsf{free}} && (|f|_{\mathsf{free}} \geq 0) \\
&= |D|_{\mathsf{free}} + |\lambda x.\overline{t}|_{\mathsf{free}} + |\lambda y.\overline{u} : \pi|_{\mathsf{free}} + |\overline{t}|_{\mathsf{free}} && (|\lambda x.\overline{t}|_{\mathsf{free}} = 0) \\
&= |s'|_{\mathsf{free}} + |\overline{t}|_{\mathsf{free}} && (\text{def. of } |s'|_{\mathsf{free}}) \\
&= |s'|_{\mathsf{free}} + |t_0| && (\text{Lemma 13}) \\
&\leq |t_0|_{\mathsf{free}} + |t_0| \cdot |\sigma|_\beta - |\sigma|_{\mathsf{s}} + |t_0| && (\textit{i.h.}) \\
&= |t_0|_{\mathsf{free}} + |t_0| \cdot (|\sigma|_\beta + 1) - |\sigma|_{\mathsf{s}} \\
&= |t_0|_{\mathsf{free}} + |t_0| \cdot |\rho|_\beta - |\rho|_{\mathsf{s}}
\end{aligned}$$

– *the last transition is a commutative transition.* Note that (sub)terms and stacks are moved around but never erased, never duplicated, and never modified. Moreover no new pieces of code are introduced, so that the measure never changes. Since also $|\rho|_\beta$ and $|\rho|_{\mathsf{s}}$ do not change, the statement follows from the *i.h.* □

Corollary 15 (Bilinear Number of Substitution Transitions). *Let $\rho : t_0^\circ \rightsquigarrow^* s$ be a Easy GLAMOUr execution. Then $|\rho|_{\mathsf{s}} \leq (1 + |\rho|_\beta) \cdot |t_0|$.* See p. 13

Proof. By Lemma 14, $|\rho|_{\mathsf{s}} \leq |t_0|_{\mathsf{free}} + |t_0| \cdot |\rho|_\beta - |s|_{\mathsf{free}}$, that implies $|\rho|_{\mathsf{s}} \leq |t_0|_{\mathsf{free}} + |t_0| \cdot |\rho|_\beta$. The statement follows from the fact that $|t_0|_{\mathsf{free}} \leq |t_0|$. □

Lemma 16 (Number of Commutative Transitions)**.** *For $\rho : t_0^\circ \rightsquigarrow^* s$ be an Easy GLAMOUr execution. Then $|\rho|_{\mathsf{c}} \leq |\rho|_{\mathsf{c}} + |s|_{\mathsf{c}} \leq (1+|\rho|_{\mathsf{s}}) \cdot |t_0| \in O((1+|\rho|_\beta) \cdot |t_0|^2)$.* See p. 13

Proof. First, note that $|\rho|_{\mathsf{c}} \leq |\rho|_{\mathsf{c}} + |s|_{\mathsf{c}}$ since $|s|_{\mathsf{c}} \geq 0$. We prove that $|\rho|_{\mathsf{c}} + |s|_{\mathsf{c}} \leq (1 + |\rho|_{\mathsf{s}}) \cdot |t_0|$ by induction on the length of the execution ρ.

Base case (empty execution): then, $t_0^\circ = s$ and $|\rho|_{\mathsf{c}} = 0 = |\rho|_{\mathsf{s}}$, thus the property collapses on the tautology $|\bar{t}_0| \leq |t_0|$.

Inductive case: let $s' \rightsquigarrow s$ be the last transition of ρ and let σ be the prefix of ρ ending on s'. The statement holds for s' by the *i.h.*, *i.e.* $|\sigma|_{\mathsf{c}} + |s'|_{\mathsf{c}} \leq (1+|\sigma|_{\mathsf{s}}) \cdot |t_0|$. We now show that the statement hold by analyzing the various cases of $s' \rightsquigarrow s$ and showing that the inequality holds also after the transition:

– *Commutative Transitions* $\rightsquigarrow_{\mathsf{c}_1}$: the rule splits the code $\overline{tu}$ between the dump and the code. Therefore, $|s|_{\mathsf{c}} = |s'|_{\mathsf{c}} - 1$ while clearly $|\rho|_{\mathsf{c}} = |\sigma|_{\mathsf{c}} + 1$, that is the lhs does not change. The rhs does not change either, and so the inequality is preserved.
– *Commutative Transitions* $\rightsquigarrow_{\mathsf{c}_{2,3}}$: these rules consume the current code, so $|s|_{\mathsf{c}} \leq |s'|_{\mathsf{c}} - 1$. Since clearly $|\rho|_{\mathsf{c}} = |\sigma|_{\mathsf{c}} + 1$, it follows that the lhs either decreases or stays the same. The rhs does not change either, and so the inequality is preserved.
– *β-Transition* $\rightsquigarrow_\beta$: trivial, as the lhs decreases of 1 (because the λ of the abstraction is consumed) and the rhs does not change.
– *Substitution Transition* $\rightsquigarrow_{\mathsf{s}}$: it modifies the current code by replacing a variable (of size 1) with an abstraction coming from the environment. Because of the subterm invariant (Lemma 13), the abstraction is a subterm of t_0 and so the increment of the lhs is bounded by $|t_0|$. We have $|\rho|_{\mathsf{s}} = |\sigma|_{\mathsf{s}} + 1$ and so the rhs increases of $|t_0|$, that is, the inequality still holds.

This ends the proof of $|\rho|_{\mathsf{c}} + |s|_{\mathsf{c}} \leq (1 + |\rho|_{\mathsf{s}}) \cdot |t_0|$. Now, substituting the bound given by Cor. 15 into $|\rho|_{\mathsf{c}} + |s|_{\mathsf{c}} \leq (1 + |\rho|_{\mathsf{s}}) \cdot |t_0|$ we obtain

$$|\rho|_{\mathsf{c}} + |s|_{\mathsf{c}} \leq (1 + |\rho|_{\mathsf{s}}) \cdot |t_0| \leq (1 + (1 + |\rho|_\beta) \cdot |t_0|) \cdot |t_0| = (1 + |\rho|_\beta) \cdot |t_0|^2 + |t_0|$$

Then $|\rho|_{\mathsf{c}} + |s|_{\mathsf{c}}$, and thus $|\rho|_{\mathsf{c}}$, is bound by $O((1 + |\rho|_\beta) \cdot |t_0|^2)$. □

See p. 14 **Theorem 17** (Easy GLAMOUr Overhead Bound)**.** *Let $\rho : t_0^\circ \rightsquigarrow^* s$ be a 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 implemented by ρ) and quadratic in the size of the initial term t_0.*

Proof. The cost of implementing ρ is the sum of the costs of implementing the β, substitution, and commutative transitions:

1. *β-Transition* $\rightsquigarrow_\beta$: each one costs $O(1)$ and so all together they cost $O(|\rho|_\beta)$.
2. *Substitution Transition* $\rightsquigarrow_{\mathsf{s}}$: by Cor. 15 we have $|\rho|_{\mathsf{s}} \leq (1 + |\rho|_\beta) \cdot |t_0|$, *i.e.* the number of substitution transitions is bilinear. By the subterm invariant (Lemma 13), each substitution step costs at most $O(|t_0|)$, and so their full cost is $O((1 + |\rho|_\beta) \cdot |t_0|^2)$.

3. *Commutative Transitions* $\rightsquigarrow_{\mathtt{c}}$: by Lemma 16 we have $|\rho|_{\mathtt{c}} \leq (1+|\rho|_{\mathtt{s}}) \cdot |t_0|$. Now, substituting the bound given by Cor. 15 we obtain

$$|\rho|_{\mathtt{c}} \leq (1+|\rho|_{\mathtt{s}}) \cdot |t_0| \leq (1+(1+|\rho|_\beta) \cdot |t_0|) \cdot |t_0| = (1+|\rho|_\beta) \cdot |t_0|^2 + |t_0|$$

Since every commutative transition evidently takes constant time, the whole cost of the commutative transitions is bound by $O((1+|\rho|_\beta) \cdot |t_0|^2)$.

Then the cost of implementing ρ is $O((1+|\rho|_\beta) \cdot |t_0|^2)$. □

B.5 Proofs of Section 6 (Fast GLAMOUr)

For the Fast GLAMOUr we proceed like for the Easy GLAMOUr: first we prove the invariants, and then we use them to prove that it forms an implementation system with respect to right-to-left evaluation $\rightarrow_{\mathtt{r}\beta_f}$ in the fireball calculus (via the decoding). The differences are minimal, but we include detailed proofs for the sake of completeness.

Lemma 32 (Fast GLAMOUr Invariants). *Let $s = (D, \overline{t}, \pi, E)$ be a reachable state. Then:*

1. Name:
   1. Explicit Substitutions*: if $E = E'[x{\leftarrow}\overline{u}]E''$ then x is fresh wrt $\overline{u}$ and E'';*
   2. Abstractions*: if $\lambda x.\overline{u}$ is a subterm of D, $\overline{u}$, π, or E then 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 either $E(x) = \bot$ or $E(x) = y@\pi''$,*
   – *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;*

Proof. By induction on the length of the execution leading to the reachable state. In an initial state all the invariants trivially hold. For a non-empty execution the proof for every invariant is by case analysis on the last transition, using the *i.h.*.

1. *Name.* Cases:
   (a) $s' = (D, \overline{tu}, \pi, E) \rightsquigarrow_{\mathtt{c}_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. Both points follow immediately from the *i.h.*
   (b) $s' = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{\mathtt{c}_2} (D, \overline{t}, \lambda x.\overline{u}@\epsilon : \pi, E) = s$. Both points follow immediately from the *i.h.*
   (c) $s' = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{\mathtt{c}_3} (D, \overline{t}, x@\pi' : \pi, E) = s$ with $E(x) = \bot$ or $E(x) = y@\pi''$ or ($E(x) = \lambda y.\overline{u}@\epsilon$ and $\pi' = \epsilon$). Both points follow immediately from the *i.h.*
   (d) $s' = (D, \lambda x.\overline{t}, y@\epsilon : \pi, E) \rightsquigarrow_{\beta_1} (D, \overline{t}\{x{\leftarrow}y\}, \pi, E) = s$. Both points follow immediately from the *i.h.*
   (e) $s' = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta_2} (D, \overline{t}, \pi, [x{\leftarrow}\phi]E) = s$ with $\phi \neq y@\epsilon$. Point 1 for the new entry in the environment follows from the *i.h.* for Point 2, for the other entries from the *i.h.* for Point 1. Point 2 follows from its *i.h.*

(f)
$$\begin{aligned} s' &= (D, x, \phi : \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2) \\ &\rightsquigarrow_{\mathtt{s}} (D, (\lambda y.\overline{u})^{\alpha}, \phi : \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2) = s. \end{aligned}$$

Point 1 follows from its *i.h.*. Point 2 for the new code is guaranteed by the α-renaming operation $(\lambda y.\overline{u})^{\alpha}$, the rest follows from its *i.h.*

2. *Fireball Item.* Cases:
   (a) $s' = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{\mathtt{c}_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. It follows from the *i.h.*
   (b) $s' = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{\mathtt{c}_2} (D, \overline{t}, \lambda x.\overline{u}@\epsilon : \pi, E) = s$. For $\lambda x.\overline{u}@\epsilon$ we have that $\underline{\lambda x.\overline{u}@\epsilon}$ and $\underline{\lambda x.\overline{u}@\epsilon}{\downarrow_E} = (\lambda x.\overline{u}){\downarrow_E} = \lambda x.\overline{u}{\downarrow_E}$ are abstractions, and hence fireballs. For all other items the invariant follows from the *i.h.*
   (c) $s' = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{\mathtt{c}_3} (D, \overline{t}, x@\pi' : \pi, E) = s$ with $E(x) = \bot$ or $E(x) = y@\pi''$ or ($E(x) = \lambda y.\overline{u}@\epsilon$ and $\pi' = \epsilon$). For $x@\pi'$, we have that $\underline{x@\pi'} = \langle x\rangle\underline{\pi'}$ and $\underline{x@\pi'}{\downarrow_E} = \langle x{\downarrow_E}\rangle(\underline{\pi'}{\downarrow_E})$. By *i.h.*, $\underline{\phi'}$ is a fireball for every item ϕ' in π'. Therefore, $\underline{x@\pi'}$ is an inert term. Concerning $\underline{x@\pi'}{\downarrow_E}$, there are three subcases:
      i. $E(x) = y@\pi''$ *i.e.* $E \coloneqq E_1[x{\leftarrow}y@\pi'']E_2$. By Lemma 32.1.1, every ES in E binds a different variable, so $x{\downarrow_E} = x{\downarrow_{E_1[x\leftarrow y@\pi'']E_2}} = x{\downarrow_{E_1}}\{x{\leftarrow}\underline{y@\pi''}\}{\downarrow_{E_2}} = \underline{y@\pi''}{\downarrow_{E_2}} = \underline{y@\pi''}{\downarrow_E}$, that by *i.h.* is an inert term. Moreover, the *i.h.* also gives that $\phi'{\downarrow_E}$ is a fireball for every item ϕ' in π'. Therefore $\underline{x@\pi'}{\downarrow_E} = \langle x{\downarrow_E}\rangle(\underline{\pi'}{\downarrow_E})$ is an inert term.
      ii. $E(x) = \bot$. Similar to the previous case. By hypothesis, we have $x{\downarrow_E} = x$. As before, by *i.h.* $\underline{\phi'}{\downarrow_E}$ is a fireball for every item ϕ' in π'. So, $\underline{x@\pi'}{\downarrow_E} = \langle x{\downarrow_E}\rangle(\underline{\pi'}{\downarrow_E}) = \langle x\rangle(\underline{\pi'}{\downarrow_E})$ is an inert term.
      iii. $E(x) = \lambda y.\overline{u}@\epsilon$ (*i.e.* $E = E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2$) and $\pi' = \epsilon$. Then $\underline{x@\pi'} = x$. By Lemma 32.1.1, every ES in E binds a different variable, so $x{\downarrow_E} = x{\downarrow_{E_1[x\leftarrow\lambda y.\overline{u}@\epsilon]E_2}} = x{\downarrow_{E_1}}\{x{\leftarrow}\underline{\lambda y.\overline{u}@\epsilon}\}{\downarrow_{E_2}} = \underline{\lambda y.\overline{u}@\epsilon}{\downarrow_{E_2}} = \lambda y.\overline{u}{\downarrow_E}$. Therefore $\underline{x@\pi'}{\downarrow_E} = x{\downarrow_E} = \lambda y.\overline{u}{\downarrow_E}$ is an abstraction.

      For all other items in s the invariant follows from the *i.h.*
   (d) $s' = (D, \lambda x.\overline{t}, y@\epsilon : \pi, E) \rightsquigarrow_{\beta_1} (D, \overline{t}\{x{\leftarrow}y\}, \pi, E) = s$. Then the invariant follows immediately from the *i.h.*
   (e) $s' = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta_2} (D, \overline{t}, \pi, [x{\leftarrow}\phi]E) = s$ with $\phi \neq y@\epsilon$. By Lemma 32.1.2 x may occur only in $\overline{t}$. Thus the substitution ${\downarrow_{[x\leftarrow\phi]E}}$ acts exactly as ${\downarrow_E}$ on every item in s. Then the invariant follows from the *i.h.*
   (f)
$$\begin{aligned} s' &= (D, x, \phi : \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2) \\ &\rightsquigarrow_{\mathtt{s}} (D, (\lambda y.\overline{u})^{\alpha}, \phi : \pi, E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2) = s. \end{aligned}$$

   It follows from the *i.h.*
3. *Contextual Decoding.* Cases:
   (a) $s' = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{\mathtt{c}_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. By *i.h.* $C_{s'} = \underline{D}\langle\underline{\pi}\rangle{\downarrow_E}$ is a right context, as well as $\overline{u}{\downarrow_E}\langle\cdot\rangle$. Then their composition $(\underline{D}\langle\underline{\pi}\rangle{\downarrow_E})\langle\overline{u}{\downarrow_E}\langle\cdot\rangle\rangle = \underline{D}\langle\langle\overline{u}\langle\cdot\rangle\rangle\underline{\pi}\rangle{\downarrow_E} = C_s$ is a right context.

(b) $s' = (D : \bar t\Diamond\pi, \lambda x.\bar u, \epsilon, E) \rightsquigarrow_{\mathtt{c}_2} (D, \bar t, \lambda x.\bar u@\epsilon : \pi, E) = s$. By *i.h.* $C_{s'} = \underline{D : \bar t\Diamond\pi}\downarrow_E = \underline{D}\langle\langle \bar t\langle\cdot\rangle\rangle\underline{\pi}\rangle\downarrow_E$ is a right context, that implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well. So, $C_{s'} = \underline{D}\langle\underline{\lambda x.\bar u@\epsilon : \pi}\rangle\downarrow_E = \underline{D}\langle\langle\langle\cdot\rangle\lambda x.\bar u\rangle\underline{\pi}\rangle\downarrow_E$ $= (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\langle\cdot\rangle\lambda x.\bar u\downarrow_E\rangle$ is a right context, because it is the composition of right context, given that $\lambda x.\bar u\downarrow_E$ is a fireball.

(c) $s' = (D : \bar t\Diamond\pi, x, \pi', E) \rightsquigarrow_{\mathtt{c}_3} (D, \bar t, x@\pi' : \pi, E) = s$ with $E(x) = \bot$ or $E(x) = y@\pi''$ or ($E(x) = \lambda y.\bar u@\epsilon$ and $\pi' = \epsilon$). By *i.h.* $C_{s'} = \underline{D : \bar t\Diamond\pi}\langle\underline{\pi'}\rangle\downarrow_E = \underline{D}\langle\langle\bar t(\underline{\pi'})\rangle\underline{\pi}\rangle\downarrow_E$ is a right context, that implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well. Then $C_s = \underline{D}\langle\underline{x@\pi' : \pi}\rangle\downarrow_E =$ $\underline{D}\langle\langle\langle\cdot\rangle\underline{x@\pi'}\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\langle\cdot\rangle\underline{x@\pi'}\downarrow_E\rangle$ is a right context, because it is the composition of right context, given that $\underline{x@\pi'}\downarrow_E$ is a fireball by Lemma 32.2.

(d) $s' = (D, \lambda x.\bar t, y@\epsilon : \pi, E) \rightsquigarrow_{\beta_1} (D, \bar t\{x{\leftarrow}y\}, \pi, E) = s$. By the *i.h.* $C_{s'} = \underline{D}\langle\underline{y@\epsilon : \pi}\rangle\downarrow_E$ is a right context, that implies that $C_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well.

(e) $s' = (D, \lambda x.\bar t, \phi : \pi, E) \rightsquigarrow_{\beta_2} (D, \bar t, \pi, [x{\leftarrow}\phi]E) = s$ with $\phi \neq y@\pi'$. By the *i.h.* $C_{s'} = \underline{D}\langle\underline{\phi : \pi}\rangle\downarrow_E$ is a right context, that implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well. Now, note that $C_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_{[x\leftarrow\phi]E} = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ because by Lemma 32.1.2 x may occur only in $\bar t$, and so the substitution $\downarrow_{[x\leftarrow\phi]E}$ acts on every code in D and π exactly as $\downarrow_E$.

(f)
$$\begin{aligned} s' &= (D, x, \phi : \pi, E_1[x{\leftarrow}\lambda y.\bar u@\epsilon]E_2) \\ &\rightsquigarrow_{\mathtt{s}} (D, (\lambda y.\bar u)^\alpha, \phi : \pi, E_1[x{\leftarrow}\lambda y.\bar u@\epsilon]E_2) = s. \end{aligned}$$

It follows by the *i.h.* because $C_{s'} = C_s$, as the only component that changes is the code. □

Lemma 33 (Fast GLAMOUr β-Projection and Overhead Transparency). *Let s be a reachable state.*

1. Overhead Transparency*: if $s \rightsquigarrow_{\mathtt{s},\mathtt{c}_{1,2,3}} s'$ then $\underline{s} = \underline{s'}$;*
2. β-Projection*: if $s \rightsquigarrow_{\beta_{1,2}} s'$ then $\underline{s} \rightarrow_{\mathtt{r}\beta_f} \underline{s'}$.*

Proof. Transitions:

1. $s = (D, \bar t\bar u, \pi, E) \rightsquigarrow_{\mathtt{c}_1} (D : \bar t\Diamond\pi, \bar u, \epsilon, E) = s'$. Then
$$\begin{aligned} \underline{s} &= \underline{D}\langle\langle\underline{\bar t\bar u}\rangle\underline{\pi}\rangle\downarrow_E \\ &= \underline{D : \bar t\Diamond\pi}\langle\underline{\bar u}\rangle\downarrow_E \\ &= \underline{D : \bar t\Diamond\pi}\langle\langle\underline{\bar u}\rangle\underline{\epsilon}\rangle\downarrow_E = \underline{s'} \end{aligned}$$

2. $s = (D : \bar t\Diamond\pi, \lambda x.\bar u, \epsilon, E) \rightsquigarrow_{\mathtt{c}_2} (D, \bar t, \lambda x.\bar u@\epsilon : \pi, E) = s'$. Then
$$\begin{aligned} \underline{s} &= \underline{D : \bar t\Diamond\pi}\langle\langle\lambda x.\bar u\rangle\underline{\epsilon}\rangle\downarrow_E \\ &= \underline{D}\langle\langle\underline{\bar t}(\langle\lambda x.\bar u\rangle\underline{\epsilon})\rangle\underline{\pi}\rangle\downarrow_E \\ &= \underline{D}\langle\langle\underline{\bar t}\rangle\underline{\lambda x.\bar u@\epsilon : \pi}\rangle\downarrow_E = \underline{s'} \end{aligned}$$

3. $s = (D : \bar{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{\mathtt{c}_3} (D, \bar{t}, x@\pi' : \pi, E) = s'$ with $E(x) = \bot$ or $E(x) = y@\pi''$ or ($E(x) = \lambda y.\overline{u}@\epsilon$ and $\pi' = \epsilon$). Then

$$\begin{aligned} \underline{s} &= \underline{D : \bar{t}\Diamond\pi}\langle\langle x\rangle\underline{\pi'}\rangle\downarrow_E \\ &= \underline{D}\langle\langle\bar{t}(\langle x\rangle\underline{\pi'})\rangle\underline{\pi}\rangle\downarrow_E \\ &= \underline{D}\langle\langle\bar{t}\rangle\underline{x@\pi' : \pi}\rangle\downarrow_E = \underline{s'} \end{aligned}$$

4. $s = (D, \lambda x.\bar{t}, y@\epsilon : \pi, E) \rightsquigarrow_{\beta_1} (D, \bar{t}\{x{\leftarrow}y\}, \pi, E) = s'$. Then

$$\begin{aligned} \underline{s} &= \underline{D}\langle\langle\lambda x.\bar{t}\rangle\underline{y@\epsilon : \pi}\rangle\downarrow_E \\ &= \underline{D}\langle\langle(\lambda x.\bar{t})y\rangle\underline{\pi}\rangle\downarrow_E \\ &\to_{\mathtt{r}\beta_f} \underline{D}\langle\langle\bar{t}\{x{\leftarrow}y\}\rangle\underline{\pi}\rangle\downarrow_E \qquad = \underline{s'} \end{aligned}$$

where the rewriting step takes place because $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context by Lemma 11.4.

5. $s = (D, \lambda x.\bar{t}, \phi : \pi, E) \rightsquigarrow_{\beta_2} (D, \bar{t}, \pi, [x{\leftarrow}\phi]E) = s'$ with $\phi \neq y@\epsilon$. Then

$$\begin{aligned} \underline{s} &= \underline{D}\langle\langle\lambda x.\bar{t}\rangle\underline{\phi : \pi}\rangle\downarrow_E \\ &= \underline{D}\langle\langle(\lambda x.\bar{t})\underline{\phi}\rangle\underline{\pi}\rangle\downarrow_E \\ &\to_{\mathtt{r}\beta_f} \underline{D}\langle\langle\bar{t}\{x{\leftarrow}\underline{\phi}\}\rangle\underline{\pi}\rangle\downarrow_E \\ &= \underline{D}\langle\langle\bar{t}\rangle\underline{\pi}\rangle\{x{\leftarrow}\underline{\phi}\}\downarrow_E \\ &= D\langle\bar{t}\rangle\underline{\pi}\downarrow_{[x\leftarrow\phi]E} \qquad = \underline{s'} \end{aligned}$$

where the rewriting step takes place because
(a) $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context by Lemma 11.4;
(b) $\underline{\phi}$ is a fireball by Lemma 11.3.
Moreover, the meta-level substitution $\{x{\leftarrow}\underline{\phi}\}$ can be extruded (in the equality step after the rewriting) without renaming x, because by Lemma 11.1.2 x does not occur in D nor π.

6. $s = (D, x, \phi{:}\pi, E) \rightsquigarrow_{\mathtt{s}} (D, (\lambda y.\overline{u})^{\alpha}, \phi{:}\pi, E) = s'$ with $E = E_1[x{\leftarrow}\lambda y.\overline{u}@\epsilon]E_2$. Then

$$\begin{aligned} \underline{s} &= \underline{D}\langle\langle x\rangle\underline{\phi : \pi}\rangle\downarrow_E \\ &= \underline{D\downarrow_E}\langle\langle x\downarrow_E\rangle\underline{\phi : \pi\downarrow_E}\rangle \\ &= \underline{D\downarrow_E}\langle\langle\lambda y.\overline{u}\downarrow_E\rangle\underline{\phi : \pi\downarrow_E}\rangle \\ &= \underline{D}\langle\langle\lambda y.\overline{u}\rangle\underline{\phi : \pi}\rangle\downarrow_E \qquad = \underline{s'} \end{aligned}$$

□

Lemma 34 (Fast GLAMOUr Progress). *Let s be a reachable final state. Then $\underline{s}$ is a fireball, i.e. it is β_f-normal.*

Proof. An immediate inspection of the transitions shows that in a final state the code cannot be an application and the dump is necessarily empty. In fact, final states have one of the following two shapes:

1. *Top-Level Unapplied Abstraction*, i.e. $s = (\epsilon, \lambda x.\bar{t}, \epsilon, E)$. Then $\underline{s} = (\lambda x.\bar{t})\downarrow_E = \lambda x.\bar{t}\downarrow_E$ that is a fireball.

2. *Top-Level Free Variable, i.e.* $s = (\epsilon, x, \epsilon, E)$ (note that, differently from what happens in the Easy GLAMOUr Machine, it might be that $E(x) \neq \bot$). We claim that $\underline{s} = x{\downarrow}_E$ is a fireball. Indeed, according to the fireball invariant item (Lemma 32.2), $\underline{\phi}$ is a fireball for any item ϕ in E; thus, the only possibility to have $x{\downarrow}_E$ different from a fireball is that there is an item ϕ in E such that $\underline{\phi}{\downarrow}_E$ is not a fireball, but this is impossible by Lemma 32.2.
3. *Top-Level Compound Inert Term, i.e.* $s = (\epsilon, x, \phi{:}\pi, E)$ with $E(x) \neq \lambda y.\bar{t}@\epsilon$. Subcases:
   (a) $E(x) = \bot$. Then $\underline{s} = (\langle x\rangle\underline{\pi}){\downarrow}_E = \langle x{\downarrow}_E\rangle(\underline{\pi}{\downarrow}_E) = \langle x\rangle(\underline{\pi}{\downarrow}_E)$. Now, by the fireball item invariant (Lemma 32.2) every element of $\underline{\pi}{\downarrow}_E$ is a fireball, so $\langle x\rangle(\underline{\pi}{\downarrow}_E)$ is an inert term, and hence a fireball.
   (b) $E(x) = y@\pi'$. Then $\underline{s} = (\langle x\rangle\underline{\pi}){\downarrow}_E = \langle x{\downarrow}_E\rangle(\underline{\pi}{\downarrow}_E) = \langle\langle y\rangle(\underline{\pi'}{\downarrow}_E)\rangle(\underline{\pi}{\downarrow}_E)$. By the fireball item invariant (Lemma 32.2), any element of $\underline{\pi}{\downarrow}_E$ and $\underline{\pi'}{\downarrow}_E$ is a fireball, so $\langle\langle y\rangle(\underline{\pi'}{\downarrow}_E)\rangle(\underline{\pi}{\downarrow}_E)$ is an inert term, and hence a fireball. □

Theorem 19 (Fast GLAMOUr Implementation). *The Fast GLAMOUr implements right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ in λ_{fire} (via the decoding $\underline{\cdot}$).* See p. 16

Proof. According to Thm. 9, it is enough to show that the Fast GLAMOUr and the right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ and the decoding $\underline{\cdot}$ form an implementation system, *i.e.* that the five conditions in Def. 8 hold. Note that substitution ($\leadsto_{\mathsf{s}}$) and commutative ($\leadsto_{\mathsf{c}_{1,2,3}}$) transitions are considered as overhead transitions, whereas the β transitions are $\leadsto_{\beta_1}$ and $\leadsto_{\beta_2}$.
1. *β-Projection*: $s \leadsto_{\beta_{1,2}} s'$ implies $\underline{s} \to \underline{s'}$ by Lemma 33.2 (recall that $\leadsto_{\beta_{1,2}} = \leadsto_{\beta_1} \cup \leadsto_{\beta_2}$ according to Note 29).
2. *Overhead Transparency*: $s \leadsto_{\mathsf{s},\mathsf{c}_{1,2,3}} s'$ implies $\underline{s} = \underline{s'}$ by Lemma 33.1 (recall that $\leadsto_{\mathsf{s},\mathsf{c}_{1,2,3}} = \leadsto_{\mathsf{s}} \cup \leadsto_{\mathsf{c}_1} \cup \leadsto_{\mathsf{c}_2} \cup \leadsto_{\mathsf{c}_3}$ according to Note 29).
3. *Overhead Transitions Terminate*: Termination of $\leadsto_{\mathsf{s},\mathsf{c}_{1,2,3}}$ is given by forthcoming Lemma 20, which is postponed because they actually give precise complexity bounds, not just termination.
4. *Determinism*: The Fast GLAMOUr machine is deterministic, as it can be seen by an easy inspection of the transitions (see Fig. 3). Lemma 3.2 proves that $\to_{\mathtt{r}\beta_f}$ is deterministic.
5. *Progress*: Let s be a Fast GLAMOUr final states. By Lemma 34, $\underline{s}$ is a β_f-normal term, in particular it is $\to_{\mathtt{r}\beta_f}$-normal because $\to_{\mathtt{r}\beta_f} \subseteq \to_{\beta_f}$. □

Complexity Analysis of the Fast GLAMOUr. As explained in the paper, the complexity analysis of the Fast GLAMOUr is essentially trivial. Here we add a few details to convince the skeptical reader.

Lemma 20 (Number of Overhead Transitions). *Let $\rho : t_0^\circ \leadsto^* s$ be a Fast GLAMOUr execution. Then* See p. 16
1. Substitution *vs* β Transitions: $|\rho|_{\mathsf{s}} \leq |\rho|_\beta$.
2. Commutative *vs* Substitution Transitions: $|\rho|_{\mathsf{c}} \leq (1+|\rho|_{\mathsf{s}})\cdot|t_0| \leq (1+|\rho|_\beta)\cdot|t_0|$.

Proof. 1. *Substitution vs β Transitions*: since abstractions are substituted on-demand, every substitution transition is followed by a β-transition. Therefore, in an execution ρ there can be at most one substitution transition not followed by a β-transition, and so $|\rho|_{\mathtt{s}} \leq |\rho|_\beta + 1$. Now, note that executions start on initial states, *i.e.* on states with empty environments where substitution transitions are not possible. So in ρ there must be a β-transition before any other substitution transition. The $+1$ can then be removed, obtaining $|\rho|_{\mathtt{s}} \leq |\rho|_\beta$.

2. *Commutative vs Substitution Transitions*: the bound $|\rho|_{\mathtt{c}} \leq (1 + |\rho|_{\mathtt{s}}) \cdot |t_0|$, that is the same as in the Easy GLAMOUr, is obtained in exactly the same way, by using the commutative size measure defined in Sect. 5. The differences in the proof are minimal:
   - Transitions $\leadsto_{\mathtt{c}_1}$ and $\leadsto_{\mathtt{c}_2}$: no difference, because they are exactly the same transitions of the Easy GLAMOUr.
   - Transition $\leadsto_{\mathtt{c}_3}$: the transition has a side-condition more than the same transition of the Easy GLAMOUr. Than it is a sub-case, and so the bound obviously hold.
   - Transition $\leadsto_{\beta_1}$: the novelty of the transition is the renaming of the code, but it lets the size, and thus the measure, unchanged.
   - Transition $\leadsto_{\beta_2}$: a special case of $\leadsto_\beta$ of the Easy GLAMOUr.
   - Transition $\leadsto_{\mathtt{s}}$: a special case of $\leadsto_{\mathtt{s}}$ of the Easy GLAMOUr.

   Last, the inequality $(1 + |\rho|_{\mathtt{s}}) \cdot |t_0| \leq (1 + |\rho|_\beta) \cdot |t_0|$ is obtained by applying the first point of the lemma. □

See p. 16

Theorem 21 (Fast GLAMOUr Bilinear Overhead)**.** *Let $\rho : t_0^\circ \leadsto^* s$ 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 and the size of the initial term.*

Proof. The cost of implementing ρ is the sum of the costs of implementing the β, substitution, and commutative transitions:

1. *β-Transitions $\leadsto_{\beta_1}$ and $\leadsto_{\beta_2}$*: $\leadsto_{\beta_1}$ costs $O(|t_0|)$ because the code has to be renamed and by the subterm invariant the size of the code is bound by $|t_0|$. Transition $\leadsto_{\beta_1}$ instead takes constant time. In the worst case all together they cost $O(|\rho|_\beta \cdot |t_0|)$.
2. *Substitution Transition $\leadsto_{\mathtt{s}}$*: by Lemma 20 we have $|\rho|_{\mathtt{s}} \leq |\rho|_\beta$. By the subterm invariant (Lemma 13), each substitution step costs at most $O(|t_0|)$, and so their full cost is $O(|\rho|_\beta \cdot |t_0|)$.
3. *Commutative Transitions $\leadsto_{\mathtt{c}}$*: by Lemma 20 $|\rho|_{\mathtt{c}} \leq (1 + |\rho|_\beta) \cdot |t_0|$. Since every commutative transition evidently takes constant time, the whole cost of the commutative transitions is bound by $O((1 + |\rho|_\beta) \cdot |t_0|)$.

Then, the cost of implementing ρ is $O((1 + |\rho|_\beta) \cdot |t_0|)$. □

B.6 Proofs of Section 7 (Conclusions)

Here we give the proof of size explosion for the family given at the end of the paper. Let us recall the definition of the family.

Let the identity combinator be $I \coloneqq \lambda z.z$ (it can in fact be replaced by any closed abstraction). Define

$$\begin{aligned} s_1 &\coloneqq \lambda x.\lambda y.(yxx) & r_0 &\coloneqq I \\ s_{n+1} &\coloneqq \lambda x.(s_n(\lambda y.(yxx))) & r_{n+1} &\coloneqq \lambda y.(yr_nr_n) \end{aligned}$$

The size exploding family is $\{s_nI\}_{n\in\mathbb{N}}$, *i.e.* it is obtained by applying s_n to the identity $I = r_0$. The statement we are going to prove is in fact more general than the one given in the paper, it is about s_nr_m instead of just s_nI, in order to obtain a simple inductive proof.

Proposition 22 (Abstraction Size Explosion)**.** *Let $n>0$. Then $s_nr_m \rightarrow^n_{\beta_\lambda} r_{n+m}$, and in particular $s_nI \rightarrow^n_{\beta_\lambda} r_n$. Moreover, $|s_nI| = O(n)$, $|r_n| = \Omega(2^n)$, s_nI is closed, and r_n is normal.* See p. 17

Proof. By induction on $n > 0$.

The base case: $s_1r_m = \lambda x.\lambda y.(yxx)r_m \rightarrow_{\beta_\lambda} (\lambda y.(yr_mr_m)) = r_{m+1}$. The inductive case: $s_{n+1}r_m = \lambda x.(s_n(\lambda y.(yxx)))r_m \rightarrow_{\beta_\lambda} s_n(\lambda y.(yr_mr_m)) = s_nr_{m+1} \rightarrow^n_{\beta_\lambda} r_{n+m+1}$, where the second sequence is obtained by the *i.h.* The rest of the statement is immediate. ☐

The family $\{s_nI\}_{n\in\mathbb{N}}$ is interesting because no matter how one looks at it, it always explodes: if evaluation is weak (*i.e.* it does not go under abstraction) there is only one possible derivation to normal form and if it is strong (*i.e.* unrestricted) all derivations have the same length (and are permutatively equivalent). Last, note that it is an example of size explosion also for Closed CbV, because the steps are weak and the term is closed. Note why machines for Closed CbV and Open CbV are not concerned with the question of substituting abstractions on-demand: the exponential number of substitutions of abstractions required by the evaluation of the family are all substitutions under abstraction, and so closed and open machines never do them anyway.