On the Relative Usefulness of Fireballs

Beniamino Accattoli
INRIA & LIX/École Polytechnique
1 rue Honoré d'Estienne d'Orves, Palaiseau, France
Email: beniamino.accattoli@inria.fr

Claudio Sacerdoti Coen
Department of Computer Science and Engineering
University of Bologna
Mura Anteo Zamboni 7, Bologna (BO), Italy
Email: claudio.sacerdoticoen@unibo.it

***Abstract*—In CSL-LICS 2014, Accattoli and Dal Lago [1] showed that there is an implementation of the ordinary (i.e. strong, pure, call-by-name) λ-calculus into models like RAM machines which is polynomial in the number of β-steps, answering a long-standing question. The key ingredient was the use of a calculus with useful sharing, a new notion whose complexity was shown to be polynomial, but whose implementation was not explored. This paper, meant to be complementary, studies useful sharing in a call-by-value scenario and from a practical point of view. We introduce the *Fireball Calculus*, a natural extension of call-by-value to open terms, that is an intermediary step towards the strong case, and we present three results. First, we adapt useful sharing, refining the meta-theory. Then, we introduce the GLAMOUr, a simple abstract machine implementing the Fireball Calculus extended with useful sharing. Its key feature is that usefulness of a step is tested—surprisingly—in constant time. Third, we provide a further optimisation that leads to an implementation having only a linear overhead with respect to the number of β-steps.**

I. Introduction

The λ-calculus is an interesting computational model because it is machine-independent, simple to define, and it compactly models functional programming languages. Its definition has only one rule, the β rule, and no data structures. The catch is the fact that the β-rule—which by itself is Turing-complete—is not atomic. Its action, namely $(\lambda x.t)u \rightarrow_\beta t\{x{\leftarrow}u\}$, can make many copies of an arbitrarily big subprogram u. In other computational models like Turing or RAM machines, an atomic operation can only move the head on the ribbon or access a register. Is β atomic in that sense? Can one count the number of β-steps to the result and then claim that it is a reasonable bound on the complexity of the term? Intuition says no, because β can be nasty, and make the program grow at an exponential rate. This is the *size explosion problem*.

Useful Sharing: nonetheless, it is possible to take the number of β-steps as an invariant cost model, *i.e.* as a complexity measure polynomially related to RAM or Turing machines. While this was known for some notable sub-calculi [2]–[6], the first proof for the general case is a recent result by Accattoli and Dal Lago [1]. Similarly to the literature, they circumvent size explosion by factoring the problem via an intermediary model in between λ-calculus and machines. Their model is the *linear substitution calculus* (LSC) [1], [7], that is a simple λ-calculus with sharing annotations (also known as explicit substitutions) where the substitution process is decomposed in micro steps, replacing one occurrence at a time. In contrast with the literature, the general case is affected by a stronger form of size explosion, requiring an additional and sophisticated layer of sharing, called *useful sharing*. Roughly, a micro substitution step is *useful* if it contributes somehow to the creation of a β-redex, and *useless* otherwise. Useful reduction then selects only useful substitution steps, avoiding the useless ones. In [1], the Useful LSC is shown to be polynomially related to both λ-calculus (in a quadratic way) and RAM machines (with polynomial overhead, conjectured linear). It therefore follows that there is a polynomial relationship $\lambda \rightarrow$ RAM. Pictorially:

$$\begin{tikzcd}
\lambda \arrow[rr, dotted, "polynomial"] \arrow[dr, "quadratic"'] & & \text{RAM} \\
& \text{Useful LSC} \arrow[ur, "polynomial\ (linear?)"'] &
\end{tikzcd}$$

Coming back to our questions, the answer of [1] is yes, β is atomic, up to a polynomial overhead. It is natural to wonder how big this overhead is. Is β reasonably atomic? Or is the degree of the polynomial big and does the invariance result only have a theoretical value? In particular, in [1] the definition of useful steps relies on a *separate* and *global* test for usefulness, that despite being tractable might not be feasible in practice. Is there an efficient way to implement the Useful LSC? Does useful sharing—*i.e.* the avoidance of useless duplications—bring a costly overhead? This paper answers these questions. But, in order to stress the practical value of the study, it shifts to a slightly different setting.

The Fireball Calculus: we recast the problem in terms of the new *fireball calculus* (FBC), essentially the weak call-by-value λ-calculus generalised to handle open terms. It is an intermediary step towards a strong call-by-value λ-calculus, that can be seen as iterated open weak evaluation. A similar approach to strong evaluation is followed also by Grégoire and Leroy in [8]. It avoids some of the complications of the strong case, and at the same time exposes all the subtleties of dealing with open terms.

Free variables are actually formalised using a distinguished syntactic class, that of *symbols*, noted a, b, c. This approach is technically convenient because it allows restricting to closed terms, so that any variable occurrence x is bound, while still having free variables, represented as symbols.

The basic idea is that—in the presence of symbols—restricting β-redex to *fire* only in presence of values is problematic. Consider indeed the following term:

$$t := ((\lambda x.\lambda y.u)(aa))w$$

where w is normal. For the usual call-by-value operational semantics t is normal (because aa is not a value) while for theoretical reasons (see [9]–[11]) one would like to be able to fire the blocked redex, reducing to $(\lambda y.u\{x{\leftarrow}aa\})w$, so that a new redex is created and the computation can continue. According to the standard classification of redex creations due to Lévy [12], this is a creation of type 1[1].

The solution we propose here is to relax the constraint about values, allowing β-redexes to fire whenever the argument is a more general structure, a so-called *fireball*, defined recursively by extending values with *inert terms*, *i.e.* applications of symbols to fireballs. In particular, aa is inert, so that *e.g.* $t \to (\lambda y.u\{x{\leftarrow}aa\})w$, as desired.

Functional languages are usually modelled by weak and *closed* calculi, so it is natural to wonder about the practical relevance of the FBC. Applications are along two axes. On the one hand, the evaluation mechanism at work in proof assistants has to deal with open terms for comparison and unification. For instance, Grégoire and Leroy's [8], meant to improve the implementation of Coq, relies on inert terms (therein called *accumulators*). On the other hand, symbols may also be interpreted as *constructors*, meant to represent data as lists or trees. The dynamics of fireballs is in fact consistent with the way constructors are handled by Standard ML [13] and in several formalisation of core ML, as in [14]. In this paper we omit destructors, whose dynamics is orthogonal to the one of β-reduction, and we expect all results presented here to carry-over with minor changes to a calculus with destructors. Therefore firing redexes involving inert terms is also justified from a practical perspective.

The Relative Usefulness of Fireballs: as we explained, the generalisation of values to fireballs is motivated by creations of type 1 induced by the firing of inert terms. There is a subtlety, however. While *substituting* a value can create a new redex (*e.g.* as in $(\lambda x.(xI))I \to (xI)\{x{\leftarrow}I\} = II$, where I is the identity—these are called creations of type 3)— substituting a inert term can not. Said differently, duplicating inert terms is useless, and leads to size explosion. Note the tension between different needs: redexes involving inert terms have to be fired (for creations of type 1), and yet the duplication and the substitution of inert terms should be avoided (since they do not give rise to creations of type 3). We solve the tension by turning to sharing, and use the simplicity of the framework to explore the implementation of useful sharing. Both values and inert terms (*i.e.* fireballs) in argument position will trigger reduction, and both will be shared just after, but only the substitution of values might be useful, because inert terms are useless. This is what we call *the relative usefulness of fireballs*. It is also why—in contrast to Grégoire and Leroy—we do not identify fireballs and values.

[1]The reader unfamiliar with redex creations should not worry. Creations are a key concept in the study of usefulness—which is why we mention them—but for the present discussion it is enough to know that there exists two kinds of creations (type 1 and the forthcoming type 3, other types will not play a role), no expertise on creations is required.

The Result: our main result is an implementation of FBC relying on useful sharing and such that it has only a linear overhead with respect to the number of β-steps. To be precise, the overhead is *bilinear*, *i.e.* linear in the number of β-steps *and* in the size of the initial term (roughly the size of the input). The dependency from the size of the initial term is induced by the action of β on whole subterms, rather than on atomic pieces of data as in RAM or Turing machines. Therefore, β is not exactly as atomic as accessing a register or moving the head of a Turing machine, and this is the price for embracing higher-order computations. Bilinearity, however, guarantees that such a price is mild and that the number of β steps—*i.e.* of function calls in a functional program—is a faithful measure of the complexity of a program. To sum up, our answer is yes, β is also reasonably atomic.

A Recipe for Bilinearity, with Three Ingredients: our proof technique is a *tour de force* progressively combining together and adapting to the FBC three recent works involving the LSC, namely the already cited invariance of useful sharing of [1], the tight relationship with abstract machines developed by Accattoli, Barenbaum, and Mazza in [15], and the optimisation of the substitution process studied by the present authors in [16]. The next section will give an overview of these works and of how they are here combined, stressing how the proof is more than a simple stratification of techniques. In particular, it was far from evident that the orthogonal solutions introduced by these works could be successfully combined together.

This Paper: the paper is meant to be self-contained, and mostly follows a didactic style. For the first half we warm up by discussing design choices, the difficulty of the problem, and the abstract architecture. The second half focuses on the results. We also suggest reading the introductions of [1], [15], [16], as they provide intuitions about concepts that here are only hinted at. Although not essential, they will certainly soften the reading of this work. Omitted proofs are in the companion technical report [17] and related work is discussed in Sect. III.

II. A Recipe with Three Ingredients

This section gives a sketch of how the bilinear implementation is built by mixing together tools from three different studies on the LSC.

1) Useful Fireballs: we start by introducing the Useful Fireball Calculus (Useful FBC), akin to the Useful LSC, and provide the proof that the relationship FBC $\to$ Useful FBC, analogously to the arrow $\lambda \to$ Useful LSC, has a quadratic overhead. Essentially, this step provides us with the following diagram:

$$\begin{tikzcd}
\text{FBC} \arrow[dr, "quadratic"'] & & \text{RAM} \\
& \text{Useful FBC} &
\end{tikzcd}$$

We go beyond simply adapting the study of [1], as the use of evaluation contexts (typical of call-by-value scenarios) leads to the new notion of *useful evaluation context*, that simplifies the technical study of useful sharing. Another key point is the *relative usefulness of fireballs*, according to their nature: only values are properly subject to the useful discipline, *i.e.* are duplicated only when they contribute somehow to β-redexes, while inert terms are never duplicated.

2) Distilling Useful Fireballs: actually, we do not follow [1] for the study of the arrow Useful FBC $\to$ RAM. We rather refine the whole picture, by introducing a further intermediary model, an *abstract machine*, mediating between the Useful FBC and RAM. We adopt the *distillation technique* of [15], that establishes a fine-grained and modular view of abstract machines as strategies in the LSC up to a notion of *structural equivalence* on terms. The general pattern arising from [15] is that for call-by-name/value/need weak and closed calculi the abstract machine adds only a bilinear overhead with respect to the shared evaluation within the LSC:

$$\begin{tikzcd}
\lambda\text{-Calculus} \arrow[d] & \text{RAM} \\
\text{LSC} \arrow[r, "bilinear"] & \text{Abstract Machine} \arrow[u, "bilinear"']
\end{tikzcd}$$

Distilleries owe their name to the fact that the LSC retains only part of the dynamics of a machine. Roughly, it isolates the relevant part of the computation, distilling it away from the search for the next redex implemented by abstract machines. The search for the redex is mapped to a notion of structural equivalence, a particular trait of the LSC, whose key property is that it can be postponed. Additionally, the transitions implementing the search for the next redex are proved to be bilinear in those simulated by the LSC: the LSC then turns out to be a complexity-preserving abstraction of abstract machines.

The second ingredient for the recipe is then a new abstract machine, called GLAMOUr, that we prove implements the Useful FBC within a bilinear overhead. Moreover, the GLAMOUr itself can be implemented within a bilinear overhead. Therefore, we obtain the following diagram:

$$\begin{tikzcd}
\text{FBC} \arrow[r, dotted, "quadratic"] \arrow[d, "quadratic"'] & \text{RAM} \\
\text{Useful FBC} \arrow[r, "bilinear"] & \text{GLAMOUr AM} \arrow[u, "bilinear"']
\end{tikzcd}$$

This is the most interesting and original step of our study. First, it shows that distilleries are compatible with open terms and useful sharing. Second, while in [15] distilleries were mainly used to revisit machines in the literature, here the distillation principles are used to guide the design of a new abstract machine. Third, useful sharing is handled via a refinement of an ordinary abstract machine relying on a basic form of labelling. The most surprising fact is that such a labelling (together with invariants induced by the call-by-value scenario) allows a straightforward and very efficient implementation of useful sharing. While the calculus is based on *separate* and *global* tests for the usefulness of a substitution step, the labelling allows the machine to do *on-the-fly* and *local* tests, requiring only constant time (!). It then turns out that implementing usefulness is much easier than analysing it. Summing up, useful sharing is easy to implement and thus a remarkable case of a theoretically born concept with relevant practical consequences.

3) Unchaining Substitutions: at this point, it is natural to wonder if the bottleneck given by the side of the diagram FBC $\to$ Useful FBC, due to the overhead of the decomposition of substitutions, can be removed. The bound on the overhead is in fact tight, and yet the answer is yes, if one refines the actors of the play. Our previous work [16], showed that (in ordinary weak and closed settings) the quadratic overhead is due to malicious chains of *renamings*, *i.e.* of substitutions of variables for variables, and that the substitution overhead reduces to linear if the evaluation is modified so that variables are never substituted, *i.e.* if values do not include variables.

For the fireball calculus the question is tricky. First of all a disclaimer: with *variables* we refer to occurrences of bound variables and not to symbols/free variables. Now, our initial definition of the calculus will exclude variables from fireballs, but useful sharing will force us to somehow reintroduce them. Our way out is an optimised form of substitution that *unchains* renaming chains, and whose overhead is proved linear by a simple amortised analysis. Such a third ingredient is first mixed with both the Useful FBC and the GLAMOUr, obtaining the Unchaining FBC and the Unchaining GLAMOUr, and then used to prove our main result, an implementation FBC $\to$ RAM having an overhead linear in the number of β steps and in the size of the initial term:

$$\begin{tikzcd}
\text{FBC} \arrow[r, dotted, "bilinear"] \arrow[d, "linear"'] & \text{RAM} \\
\text{Unchaining FBC} \arrow[r, "bilinear"] & \text{Unchaining GLAMOUr} \arrow[u, "bilinear"']
\end{tikzcd}$$

In this step, the original content is that the unchaining optimisation—while inspired by [16]—is subtler to define than in [16], as bound variables cannot be simply removed from the definition of fireballs, because of usefulness. Moreover, we also show how such an optimisation can be implemented at the machine level.

The next section discusses related work. Then there will be a long preliminary part providing basic definitions, an abstract decomposition of the implementation, and a quick study of both a calculus, the Explicit FBC, and a machine, the Open GLAM, without useful sharing. Both the calculus and the machine will not have any good asymptotical property, but they will be simple enough to familiarise the reader with the framework and with the many involved notions.

III. Related Work

In the literature, invariance results for the weak call-by-value λ-calculus have been proved three times, independently. First, by Blelloch and Greiner [2], while studying cost models for parallel evaluation. Then by Sands, Gustavsson and Moran [3], while studying speedups for functional languages, and finally by Dal Lago and Martini [4], who addressed the invariance thesis for λ-calculus. Among them, [3] is the closest one, as it also provides an abstract machine and bounds its overhead. These works however concern closed terms, and so they deal with a much simpler case. Other simple call-by-name cases are studied in [5] and [6]. The difficult case of the strong λ-calculus has been studied in [1], which is also the only reference for useful sharing.

The LSC is a variation over a λ-calculus with ES by Robin Milner [18], [19], obtained by plugging in some of the ideas of the structural λ-calculus by Accattoli and Kesner [20], introduced as a syntactic reformulation of linear logic proof nets. The LSC is similar to calculi studied by De Bruijn [21] and Nederpelt [22]. Its first appearances are in [6], [23], but its inception is actually due to Accattoli and Kesner.

Many abstract machines can be rephrased as strategies in *λ-calculi with explicit substitutions* (ES), see at least [24]–[29].

The related work that is by far closer to ours is the already cited study by Grégoire and Leroy of an abstract machine for call-by-value weak and open reduction in [8]. We developed our setting independently, and yet the FBC is remarkably close to their calculus, in particular our *inert terms* are essentially their *accumulators*. The difference is that our work is complexity-oriented while theirs is implementation-oriented. On the one hand they do not recognise the relative usefulness of fireballs, and so their machine is not invariant, *i.e.* our machine is more efficient and on some terms even exponentially faster. On the other hand, they extend the language up to the calculus of constructions, present a compilation to bytecode, and certify in Coq the correctness of the implementation.

The abstract machines in this paper use *global* environments, an approach followed only by a minority of authors (*e.g.* [3], [15], [30], [31]) and essentially identifying the environment with a store. The distillation technique was developed to better understand the relationship between the KAM and weak linear head reduction pointed out by Danos & Regnier [32]. The idea of distinguishing between *operational content* and *search for the redex* in an abstract machine is not new, as it underlies in particular the *refocusing semantics* of Danvy and Nielsen [33]. Distilleries however bring an original refinement where logic, rewriting, and complexity enlighten the picture, leading to formal bounds on machine overheads.

Our unchaining optimisation is a lazy variant of an optimisation that repeatedly appeared in the literature, often with reference to space consumption and *space leaks*, for instance in [3] as well as in Wand's [34] (section 2), Friedman et al.'s [35] (section 4), and Sestoft's [36] (section 4).

IV. The Fireball Calculus

The setting is the one of the call-by-value λ-calculus extended with symbols a, b, c, meant to denote free variables (or constructors). The syntax is:

$$\begin{array}{llcl} \text{Terms} & t,u,w,r & ::= & x \mid a \mid \lambda x.t \mid tu \\ \text{Values} & v,v' & ::= & \lambda x.t \end{array}$$

with the usual notions of free and bound variables, capture-avoiding substitution $t\{x{\leftarrow}u\}$, and closed (*i.e.* without free variables) term. We will often restrict to consider closed terms, the idea being that an open term as $x(\lambda y.zy)$ is rather represented as the closed term $a(\lambda y.by)$.

The ordinary (*i.e.* without symbols) call-by-value λ-calculus has a nice operational characterisation of values:

closed normal forms are values

Now, the introduction of symbols breaks this property, because there are closed normal forms as $a(\lambda x.x)$ that are not values. In order to restore the situation, we generalise values to *fireballs*[2], that are either values v or *inert terms* A, *i.e.* symbols possibly applied to fireballs. Associating to the left, fireballs and inerts are compactly defined by

$$\begin{array}{llcll} \text{Fireballs} & f,g,h & ::= & v \mid A & \\ \text{Inert Terms} & A,B,C & ::= & af_1 \ldots f_n & n \geq 0 \end{array}$$

For instance, $\lambda x.y$ and a are fireballs, as well as $a(\lambda x.x)$, ab, and $(a(\lambda x.x))(bc)(\lambda y.(zy))$. Fireballs can also be defined more atomically by mixing values and inert terms as follows:

$$f \ ::= \ v \mid A \qquad\qquad A \ ::= \ a \mid Af$$

Note that AB and AA are always inert.

Next, we generalise the call-by-value rule $(\lambda x.t)v \rightarrow_{\beta_v} t\{x{\leftarrow}v\}$ to substitute fireballs f rather than values v. First of all, we define a notion of evaluation context (noted F rather than E, reserved to forthcoming global environments), mimicking right-to-left CBV evaluation:

$$\text{Evaluation Contexts} \quad F \ ::= \ \langle\cdot\rangle \mid tF \mid Ff$$

note the case Ff, that in CBV would be Fv. Last, we define the $\mathtt{f}$(fireball) rule $\rightarrow_{\mathtt{f}}$ as follows

Rule at Top Level	Contextual closure
$(\lambda x.t)f \mapsto_{\mathtt{f}} t\{x{\leftarrow}f\}$	$F\langle t\rangle \rightarrow_{\mathtt{f}} F\langle u\rangle$ if $t \mapsto_{\mathtt{f}} u$

Our definitions lead to:

Theorem 1.
1) *Closed normal forms are fireballs.*
2) *$\rightarrow_{\mathtt{f}}$ is deterministic.*

In the introduction we motivated the notion of fireball both from theoretical and practical points of view. Theorem 1.1 provides a further, strong justification: it expresses a sort of internal harmony of the FBC, allowing to see it as the canonical completion of call-by-value to the open setting.

V. Size Explosion

Size explosion is the side effect of a discrepancy between the dynamics and the representation of terms. The usual substitution $t\{x{\leftarrow}u\}$ makes copies of u for all the occurrences of x, even if u is *useless*, *i.e.* it is normal and it does not create redexes after substitution. These copies are the burden leading to the exponential growth of the size. To illustrate the problem, let's build a size exploding family of terms.

Note that a inert term A, when applied to itself still is a inert term AA. In particular, it still is a fireball, and so it can

[2]About *fireball*: the first choice was *fire-able*, but then the spell checker suggested *fireball*.

Table I
Syntax, Rewriting Rules, and Structural Equivalence of the Explicit FBC

t, u, w, r	$::=$	$x \mid a \mid \lambda x.t \mid tu \mid t[x{\leftarrow}u]$
v, v'	$::=$	$\lambda x.t$
L, L'	$::=$	$\langle\cdot\rangle \mid L[x{\leftarrow}t]$
A, B, C	$::=$	$a \mid L\langle A\rangle L\langle f\rangle$
f, g, h	$::=$	$v \mid A$
F	$::=$	$\langle\cdot\rangle \mid tF \mid FL\langle f\rangle \mid F[x{\leftarrow}t]$

Rule at Top Level	Contextual Closure
$L\langle\lambda x.t\rangle L'\langle f\rangle \mapsto_{\mathtt{m}} L\langle t[x{\leftarrow}L'\langle f\rangle]\rangle$	$F\langle t\rangle \multimap_{\mathtt{m}} F\langle u\rangle$ if $t \mapsto_{\mathtt{m}} u$
$F\langle x\rangle[x{\leftarrow}L\langle f\rangle] \mapsto_{\mathtt{e}} L\langle F\langle f\rangle[x{\leftarrow}f]\rangle$	$F\langle t\rangle \multimap_{\mathtt{e}} F\langle u\rangle$ if $t \mapsto_{\mathtt{e}} u$

$t[x{\leftarrow}u][y{\leftarrow}w]$	$\equiv_{com}$	$t[y{\leftarrow}w][x{\leftarrow}u]$	if $y \notin \mathtt{fv}(u)$ and $x \notin \mathtt{fv}(w)$
$(tw)[x{\leftarrow}u]$	$\equiv_{@r}$	$tw[x{\leftarrow}u]$	if $x \notin \mathtt{fv}(t)$
$(tw)[x{\leftarrow}u]$	$\equiv_{@l}$	$t[x{\leftarrow}u]w$	if $x \notin \mathtt{fv}(w)$
$t[x{\leftarrow}u][y{\leftarrow}w]$	$\equiv_{[\cdot]}$	$t[x{\leftarrow}u[y{\leftarrow}w]]$	if $y \notin \mathtt{fv}(t)$

be used as an argument for redexes. We can then easily build a term of size linear in n that in n steps evaluates a complete binary tree A^{2^n}. Namely, define the family of terms t_n for $n \geq 1$:

$$\begin{array}{lcl} t_1 & := & \lambda x_1.(x_1 x_1) \\ t_{n+1} & := & \lambda x_{n+1}.(t_n(x_{n+1}x_{n+1})) \end{array}$$

Now consider $t_n A$, that for a fixed A has size linear in n. The next proposition shows that $t_n A$ reduces in n steps to A^{2^n}, causing size explosion.

Proposition 1 (Size Explosion in the FBC)**.** $t_n A \to_{\mathtt{f}}^{n} A^{2^n}$.

Proof: by induction on n. Let $B := A^2 = AA$. Cases:

$$\begin{array}{lcllcl} t_1 & = & (\lambda x_1.(x_1x_1))A & \to_{\mathtt{f}} & A^2 \\ t_{n+1} & = & (\lambda x_{n+1}.(t_n(x_{n+1}x_{n+1})))A & \to_{\mathtt{f}} & \\ & & t_nA^2 = t_nB & \to_{\mathtt{f}}^{n} & (\textit{i.h.}) \\ & & B^{2^n} & = & A^{2^{n+1}} \end{array}$$ ■

VI. Fireballs and Explicit Substitutions

In a ordinary weak scenario, sharing of subterms prevents size explosion. In the FBC however this is no longer true, as we show in this section. Sharing of subterms is here represented in a variation over the Linear Substitution Calculus, a formalism with explicit substitutions coming from a linear logic interpretation of the λ-calculus. At the dynamic level, the *small-step* operational semantics of the FBC is refined into a *micro-step* one, where explicit substitutions replace one variable occurrence at a time, similarly to abstract machines.

The terms of the *Explicit Fireball Calculus* (Explicit FBC) are:

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

where $t[x{\leftarrow}u]$ is the explicit substitution (ES) of u for x in t, that is an alternative notation for `let` $x = u$ `in` t, and where x becomes bound (in t). We silently work modulo α-equivalence of these bound variables, *e.g.* $(xy)[y{\leftarrow}t]\{x{\leftarrow}y\} = (yz)[z{\leftarrow}t]$. We use $\mathtt{fv}(t)$ for the set of free variables of t.

Contexts: the dynamics of explicit substitutions is defined using (one-hole) contexts. *Weak contexts* subsume all the kinds of context in the paper, and are defined by

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

The plugging $W\langle t\rangle$ of a term t into a context W is defined as $\langle\cdot\rangle\langle t\rangle := t$, $(\lambda x.W)\langle t\rangle := \lambda x.(W\langle t\rangle)$, and so on. As usual, plugging in a context can capture variables, *e.g.* $((\langle\cdot\rangle y)[y{\leftarrow}t])\langle y\rangle = (yy)[y{\leftarrow}t]$. The plugging $W\langle W'\rangle$ of a context W' into a context W is defined analogously. Since all kinds of context we will deal with will be weak, the definition of plugging applies uniformly to all of them.

A special and frequently used class of contexts is that of *substitution contexts* $L ::= \langle\cdot\rangle \mid L[x{\leftarrow}t]$.

Switching from the FBC to the Explicit FBC the syntactic categories of *inert terms* A, *fireballs* f, and *evaluation contexts* F are generalised in Table I as to include substitution contexts L. Note that fireballs may now contain substitutions, but *not at top level*, because it is technically convenient to give a separate status to a fireball f in a substitution context L: terms of the form $L\langle f\rangle$ are called *answers*. An *initial term* is a closed term with no explicit substitutions.

Rewriting Rules: the fireball rule $\to_{\mathtt{f}}$ is replaced by $\multimap_{\mathtt{f}}$, defined as the union of the two rules $\multimap_{\mathtt{m}}$ and $\multimap_{\mathtt{e}}$ in Table I:

1) *Multiplicative* $\multimap_{\mathtt{m}}$: is a version of $\to_{\mathtt{f}}$ where $\lambda x.t$ and f can have substitution contexts L and L' around, and the substitution is delayed.
2) *Exponential* $\multimap_{\mathtt{e}}$: the substitution or exponential rule $\multimap_{\mathtt{e}}$ replaces exactly one occurrence of a variable x currently under evaluation (in F) with its definiendum f given by the substitution. Note the apparently strange position of L in the reduct. It is correct: L has to commute outside to bind both copies of f, otherwise the rule would create free variables.

The name of the rules are due to the linear logic interpretation of the LSC.

Unfolding: the shared representation is related to the usual one via the crucial notion of *unfolding*, producing the λ-term $t{\downarrow}$ denoted by t and defined by:

$$\begin{array}{lcllcl} x{\downarrow} & := & x & (tu){\downarrow} & := & t{\downarrow}u{\downarrow} \\ (\lambda x.t){\downarrow} & := & \lambda x.t{\downarrow} & t[x{\leftarrow}u]{\downarrow} & := & t{\downarrow}\{x{\leftarrow}u{\downarrow}\} \end{array}$$

Note that $r_n{\downarrow} = A^{2^n}$.

As for the FBC, evaluation is well-defined:

Theorem 2.

1) *Closed normal forms are answers, i.e. fireballs in substitution contexts.*
2) $\multimap_{\mathtt{f}}$ *is deterministic.*

Structural Equivalence: the calculus is endowed with a structural equivalence, noted $\equiv$, whose property is to be a strong bisimulation with respect to $\multimap_{\mathtt{f}}$. It is the least equivalence relation closed by weak contexts defined by the axioms in Table I.

Proposition 2 ($\equiv$ is a Strong Bisimulation wrt $\multimap_{\mathtt{f}}$)**.** *Let* $\mathtt{x} \in \{\mathtt{sm}, \mathtt{se}\}$. *Then,* $t \equiv u$ *and* $t \multimap_{\mathtt{x}} t'$ *implies that there exists* u' *such that* $u \multimap_{\mathtt{x}} u'$ *and* $t' \equiv u'$.

Size Explosion, Again: coming back to the size explosion example, the idea is that—to circumvent it—t_n should better $\multimap_{\mathtt{m}}$-evaluate to:

$$r_n := (x_0 x_0)[x_0 \leftarrow x_1^2][x_1 \leftarrow x_2^2] \ldots [x_{n-1} \leftarrow x_n^2][x_n \leftarrow A]$$

which is an alternative, compact representation of A^{2^n}, of size linear in n, and with just one occurrence of A. Without symbols, ES are enough to circumvent size explosion [2]–[4]. In our case however they fail. The evaluation we just defined indeed does not stop on the desired compact representation, and in fact a linear number of steps (namely $3n$) may still produce an exponential output (in a substitution context).

Proposition 3 (Size Explosion in the Explicit FBC)**.** $t_n A (\multimap_{\mathtt{m}} \multimap_{\mathtt{e}}^2)^n L\langle A^{2^n}\rangle$.

Proof: by induction on n. Let $B := A^2 = AA$. Cases:

$$\begin{array}{rcll} t_1 & = & (\lambda x_1.(x_1 x_1))A & \multimap_{\mathtt{m}} \\ & & (x_1 x_1)[x_1 \leftarrow A] & \multimap_{\mathtt{e}} \\ & & (x_1 A)[x_1 \leftarrow A] & \multimap_{\mathtt{e}} \\ & & (AA)[x_1 \leftarrow A] & = \quad A^2[x_1 \leftarrow A] \end{array}$$

$$\begin{array}{rcll} t_{n+1} & = & (\lambda x_{n+1}.(t_n(x_{n+1} x_{n+1})))A & \multimap_{\mathtt{m}} \multimap_{\mathtt{e}}^2 \\ & & (t_n A^2)[x_1 \leftarrow A] = L\langle t_n B\rangle & (\multimap_{\mathtt{m}} \multimap_{\mathtt{e}}^2)^n \ (i.h.) \\ & & L'\langle B^{2^n}\rangle = L'\langle A^{2^{n+1}}\rangle & \blacksquare \end{array}$$

Before introducing useful evaluation—that will liberate us from size explosion—we are going to fully set up the architecture of the problem, by explaining 1) how ES implement a calculus, 2) how an abstract machine implements a calculus with ES, and 3) how to define an abstract machine for the inefficient Explicit FBC. Only by then (Sect. XI) we will start optimising the framework, first with useful sharing and then by eliminating renaming chains.

VII. Two Levels Implementation

Here we explain how the the small-step strategy $\rightarrow_{\mathtt{f}}$ of the FBC is implemented by a micro-step strategy $\multimap$. We are looking for an appropriate strategy $\multimap$ with ES which is polynomially related to both $\rightarrow_{\mathtt{f}}$ and an abstract machine. Then we need two theorems:

1) *High-Level Implementation*: $\rightarrow_{\mathtt{f}}$ terminates iff $\multimap$ terminates. Moreover, $\rightarrow_{\mathtt{f}}$ is implemented by $\multimap$ with only a polynomial overhead. Namely, $t \multimap^k u$ iff $t \rightarrow_{\mathtt{f}}^h u{\downarrow}$ with k polynomial in h;
2) *Low-Level Implementation*: $\multimap$ is implemented on an abstract machine with an overhead in time which is polynomial in both k and the size of t.

We will actually be more accurate, giving linear or quadratic bounds, but this is the general setting.

A. High-Level Implementation

First, terminology and notations. *Derivations* $d, e, \ldots$ are sequences of rewriting steps. With $|d|$, $|d|_{\mathtt{m}}$, and $|d|_{\mathtt{e}}$ we denote respectively the length, the number of multiplicative, and exponential steps of d.

Definition 1. *Let* $\rightarrow_{\mathtt{f}}$ *be a deterministic strategy on FBC-terms and* $\multimap$ *a deterministic strategy for terms with ES. The pair* $(\rightarrow_{\mathtt{f}}, \multimap)$ *is a* ***high-level implementation system*** *if whenever* t *is a* λ*-term and* $d : t \multimap^* u$ *then:*

1) Normal Form*: if* u *is a* $\multimap$*-normal form then* $u{\downarrow}$ *is a* $\rightarrow_{\mathtt{f}}$*-normal form.*
2) Projection*:* $d{\downarrow} : t{\downarrow} \rightarrow_{\mathtt{f}}^* u{\downarrow}$ *and* $|d{\downarrow}| = |d|_{\mathtt{m}}$.

Moreover, it is

1) locally bounded*: if the length of a sequence of substitution* $\mathtt{e}$*-steps from* u *is linear in the number* $|d|_{\mathtt{m}}$ *of* $\mathtt{m}$*-steps in* d*;*
2) globally bounded*: if* $|d|_{\mathtt{e}}$ *is linear in* $|d|_{\mathtt{m}}$.

The normal form and projection properties address the *qualitative* part, *i.e.* the part about termination. The normal form property guarantees that $\multimap$ does not stop prematurely, so that when $\multimap$ terminates $\rightarrow_{\mathtt{f}}$ cannot keep going. The projection property guarantees that termination of $\rightarrow_{\mathtt{f}}$ implies termination of $\multimap$. The two properties actually state a stronger fact: $\rightarrow_{\mathtt{f}}$ *steps can be identified with the* $\multimap_{\mathtt{m}}$*-steps of the* $\multimap$ *strategy.*

The local and global bounds allow to bound the overhead introduced by the Explicit FBC wrt the FBC, because by relating $\multimap_{\mathtt{m}}$ and $\multimap_{\mathtt{e}}$ steps, they relate $|d|$ and $|d{\downarrow}|$, since $\rightarrow_{\mathtt{f}}$ and $\multimap_{\mathtt{m}}$ steps can be identified.

The high-level part can now be proved abstractly.

Theorem 3 (High-Level Implementation)**.** *Let* t *be an ordinary* λ*-term and* $(\rightarrow_{\mathtt{f}}, \multimap)$ *a high-level implementation system.*

1) Normalisation*:* t *is* $\rightarrow_{\mathtt{f}}$*-normalising iff it is* $\multimap$*-normalising.*
2) Projection*: if* $d : t \multimap^* u$ *then* $d{\downarrow} : t \rightarrow_{\mathtt{f}}^* u{\downarrow}$

Moreover, the overhead of $\multimap$ *is, depending on the system:*

1) locally bounded*: quadratic, i.e.* $|d| = O(|d{\downarrow}|^2)$.
2) globally bounded*: linear, i.e.* $|d| = O(|d{\downarrow}|)$.

Let us see our framework at work:

Theorem 4. $(\rightarrow_{\mathtt{f}}, \multimap_{\mathtt{f}})$ *is a high-level implementation system.*

Note the absence of complexity bounds. In fact, $(\rightarrow_{\mathtt{f}}, \multimap_{\mathtt{f}})$ is not even locally bounded. Let t^n here be defined by $t^1 = t$ and $t^{n+1} = t^n t$, and $u_n := (\lambda x.x^n)A$. Then $d : u_n \multimap_{\mathtt{m}} \multimap_{\mathtt{e}}^n A^n[x \leftarrow A]$ is a counter-example to local boundedness. Moreover, the Explicit FBC also suffers of size explosion, *i.e.* implementing a single step may take exponential time. In Sect. XI useful sharing will solve these issues.

B. Low-Level Implementation: Abstract Machines

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

A machine computes using transitions, whose union is noted $\rightsquigarrow$, of two types. The *principal* one, noted $\rightsquigarrow_{\mathtt{p}}$, corresponds to the firing of a rule defining $\multimap$. In doing so, the machine can differ from the calculus implemented by a transformation of the evaluation context to an equivalent one, up to structural equivalence $\equiv$. The *commutative* transitions, noted $\rightsquigarrow_{\mathtt{c}}$, implement the search for the next redex to be fired by rearranging the data-structures to single out a new evaluation context, and they are invisible on the calculus. The names reflect a proof-theoretical view, as machine transitions can be seen as cut-elimination steps [15], [29]. Garbage collection is here simply ignored, as in the LSC it can always be postponed.

To preserve correctness, structural equivelance $\equiv$ is required to commute with evaluation $\multimap$, *i.e.* to satisfy

$$\begin{tikzcd} t \arrow[r, "\multimap" description, phantom] \arrow[r] & r & & & t \arrow[r] & r \\ \equiv & & \Rightarrow \exists q \text{ s.t.} & & \equiv & \equiv \\ u & & & & u \arrow[r] & q \end{tikzcd}$$

for each of the rules of $\multimap$, preserving the kind of rule. In fact, this means that $\equiv$ is a *strong* bisimulation (*i.e. one* step to *one* step) with respect to $\multimap$. Strong bisimulations formalise transformations which are transparent with respect to the behaviour, even at the level of complexity, because they can be retarded without affecting the length of evaluation:

Lemma 1 ($\equiv$ Postponement). *If $\equiv$ is a strong bisimulation and $t\ (\to \cup \equiv)^*\ u$ then $t \to^*\equiv u$ and the number and kind of steps of $\multimap$ in the two reduction sequences is the same.*

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

Definition 2. *A* distillery $\mathtt{D} = (\mathtt{M}, \multimap, \equiv, \underline{\cdot})$ *is given by:*

1) *An* abstract machine $\mathtt{M}$*, given by*
   a) *a deterministic labeled* transition *system* $\rightsquigarrow$ *on states s;*
   b) *a distinguished class of states deemed* initial*, in bijection with closed λ-terms and from which one obtains the* reachable *states by applying* $\rightsquigarrow^*$*;*
   c) *a partition of the labels of the transition system* $\rightsquigarrow$ *as:*
      - principal *transitions, noted* $\rightsquigarrow_{\mathtt{p}}$,
      - commutative *transitions, noted* $\rightsquigarrow_{\mathtt{c}}$*;*
2) *a deterministic* strategy $\multimap$*;*
3) *a* structural equivalence $\equiv$ *on terms s.t. it is a strong bisimulation with respect to* $\multimap$*;*
4) *a* distillation $\underline{\cdot}$*, i.e. a decoding function from states to terms, s.t. on reachable states:*
   - Principal*: $s \rightsquigarrow_{\mathtt{p}} s'$ implies $\underline{s} \multimap\equiv \underline{s'}$,*
   - Commutative*: $s \rightsquigarrow_{\mathtt{c}} s'$ implies $\underline{s} \equiv \underline{s'}$.*

We will soon prove that a distillery implies a simulation theorem, but we want a stronger form of relationship. Additional hypothesis are required to obtain the converse simulation, handle explicit substitution, and talk about complexity bounds.

Some terminology first. An *execution* ρ is a sequence of transition from an initial state. With $|\rho|$, $|\rho|_p$, and $|\rho|_c$ we denote respectively the length, the number of principal, and commutative transitions of ρ. The *size* of a term is noted $|t|$.

Definition 3 (Distillation Qualities)**.** *A distillery is*

- Reflective *when on reachable states:*
  - Termination*: $\rightsquigarrow_{\mathtt{c}}$* terminates*;*
  - Progress*: if $\underline{s}$ reduces then s is not final.*
- Explicit *when*
  - Partition*: principal transitions are partitioned into* multiplicative $\rightsquigarrow_m$ *and* exponential $\rightsquigarrow_e$*, like for the strategy* $\multimap$.
  - Explicit decoding*: the partition is preserved by the decoding, i.e.*
    * Multiplicative*: $s \rightsquigarrow_m s'$ implies $\underline{s} \multimap_{\mathtt{m}}\equiv \underline{s'}$;*
    * Exponential*: $s \rightsquigarrow_e s'$ implies $\underline{s} \multimap_{\mathtt{e}}\equiv \underline{s'}$;*
- Bilinear *when it is reflective and*
  - Execution Length*: given an execution ρ from an initial term t, the number of commutative steps $|\rho|_c$ is linear in both $|t|$ and $|\rho|_p$ (with a slightly stronger dependency on $|t|$, due to the time needed to recognise a normal form), i.e. if $|\rho|_c = O((1 + |\rho|_p) \cdot |t|)$.*
  - Commutative*: $\rightsquigarrow_{\mathtt{c}}$ is implementable on RAM in a constant number of steps;*
  - Principal*: $\rightsquigarrow_{\mathtt{p}}$ is implementable on RAM in $O(|t|)$ steps.*

A reflective distillery is enough to obtain a bisimulation between the strategy $\multimap$ and the machine $\mathtt{M}$, that is strong up to structural equivalence $\equiv$. With $|\rho|_{\mathtt{m}}$ and $|\rho|_{\mathtt{e}}$ we denote respectively the number of multiplicative and exponential transitions of ρ.

Theorem 5 (Correctness and Completeness)**.** *Let $\mathtt{D}$ be a reflective distillery and s an initial state.*

1) Strong Simulation*: for every execution $\rho : s \rightsquigarrow^* s'$ there is a derivation $d : \underline{s} \multimap^*\equiv \underline{s'}$ s.t. $|\rho|_p = |d|$.*
2) Reverse Strong Simulation*: for every derivation $d : \underline{s} \multimap^* t$ there is an execution $\rho : s \rightsquigarrow^* s'$ s.t. $t \equiv \underline{s'}$ and $|\rho|_p = |d|$.*

Moreover, if $\mathtt{D}$ is explicit then $|\rho|_{\mathtt{m}} = |d|_{\mathtt{m}}$ and $|\rho|_{\mathtt{e}} = |d|_{\mathtt{e}}$.

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

Theorem 6 (Low-Level Implementation Theorem)**.** *Let $\multimap$ be a strategy on terms with ES and $\mathtt{D} = (\mathtt{M}, \multimap, \equiv, \underline{\cdot})$ a bilinear distillery. Then a $\multimap$-derivation d is implementable on RAM machines in $O((1 + |d|) \cdot |t|)$ steps, i.e. bilinear in the size of the initial term t and the length of the derivation $|d|$.*

Proof: given $d : t \multimap^n u$ by Theorem 5.2 there is an execution $\rho : s \rightsquigarrow^* s'$ s.t. $u \equiv \underline{s'}$ and $|\rho|_p = |d|$. The number

Table II
OPEN GLAM: DATA-STRUCTURES, DECODING AND TRANSITIONS

$$\begin{array}{rcl@{\qquad}rcl}
\phi & ::= & \overline{t} \mid (\overline{t},\pi) & E,E' & ::= & \epsilon \mid [x{\leftarrow}\overline{t}] : E \\
\pi,\pi' & ::= & \epsilon \mid \phi : \pi & s,s' & ::= & (D,\overline{t},\pi,E) \\
D,D' & ::= & \epsilon \mid D : (\overline{t},\pi) & & &
\end{array}
\quad\Bigg|\quad
\begin{array}{rcl@{\qquad}rcl}
\underline{\epsilon} & := & \langle\cdot\rangle & \underline{[x{\leftarrow}\overline{t}] : E} & := & \langle\langle\cdot\rangle[x{\leftarrow}\overline{t}]\rangle\underline{E} \\
\underline{\phi : \pi} & := & \langle\langle\cdot\rangle\underline{\phi}\rangle\underline{\pi} & \underline{F_s} & := & \langle\underline{D}\langle\underline{\pi}\rangle\rangle\underline{E} \\
\underline{(t,\pi)} & := & \langle t\rangle\underline{\pi} & \underline{s} & := & \underline{F_s}\langle t\rangle \\
\underline{D : (\overline{t},\pi)} & := & \underline{D}\langle\langle\overline{t}\langle\rangle\rangle\underline{\pi}\rangle & \multicolumn{3}{l}{\text{where } s = (D,\overline{t},\pi,E)}
\end{array}$$

D	$\overline{t}\overline{u}$	π	E	$\rightsquigarrow_{\mathtt{c}_1}$	$D : (\overline{t},\pi)$	$\overline{u}$	ϵ	E
D	$\lambda x.\overline{t}$	$\overline{u} : \pi$	E	$\rightsquigarrow_{\mathtt{m}}$	D	$\overline{t}$	π	$[x{\leftarrow}\overline{u}]E$
$D : (\overline{t},\pi)$	$\lambda x.\overline{u}$	ϵ	E	$\rightsquigarrow_{\mathtt{c}_2}$	D	$\overline{t}$	$\lambda x.\overline{u} : \pi$	E
$D : (\overline{t},\pi)$	a	π'	E	$\rightsquigarrow_{\mathtt{c}_3}$	D	$\overline{t}$	$(a,\pi') : \pi$	E
D	x	π	$E_1[x{\leftarrow}\overline{u}]E_2$	$\rightsquigarrow_{\mathtt{e}}$	D	$\overline{u}^{\alpha}$	π	$E_1[x{\leftarrow}\overline{u}]E_2$

where $\overline{u}^{\alpha}$ is any code α-equivalent to $\overline{u}$ that preserves well-naming of the machine, i.e. such that any bound name in $\overline{u}^{\alpha}$ is fresh with respect to those in D, π and $E_1[x{\leftarrow}\overline{u}]E_2$.

of RAM steps to implement ρ is the sum of the number for the commutative and the principal transitions. By bilinearity, $|\rho|_c = O((1+|\rho|_p)\cdot|t|)$ and so all the commutative transitions in ρ require $O((1+|\rho|_p)\cdot|t|)$ steps, because a single one takes a constant number of steps. Again by bilinearity, each principal one requires $O(|t|)$ steps, and so all the principal transitions together require $O(|\rho|_p \cdot |t|)$ steps. ∎

We will discuss three distilleries, summarised in Table IV (page 11), and two of them will be bilinear. The machines will be sophisticated, so that we will first present a machine for the inefficient Explicit FBC (Sect. VIII, called Open GLAM), that we will later refine with useful sharing (Sect. XII, GLAMOUr) and with renaming chains elimination (Sect. XIV, Unchaining GLAMOUr).

Let us point out an apparent discrepancy with the literature. For the simpler case without symbols, the number of commutative steps of the abstract machine studied in [3] is truly linear (and not bilinear), *i.e.* it does not dependent on the size of the initial term. Three remarks:

1) *Complete Evaluation*: it is true only for evaluation to normal form, while our theorems are also valid for prefixes of the evaluation and diverging evaluations.
2) *Normal Form Recognition*: it relies on the fact that closed normal forms (*i.e.* values) can be recognised in constant time, by simply checking the topmost constructor. With symbols checking if a term is normal requires time linear in its size; linearity is simply not possible.
3) *Asymptotically Irrelevant*: the dependency from the initial term disappears from the number of commutative transitions but still affects the cost of the principal ones, because every exponentials transition copies a subterm of the initial term, and thus it takes $O(|t|)$ time.

VIII. AN INEFFICIENT DISTILLERY: THE OPEN GLAM

In this section we introduce the Open GLAM machine and show that it distills to the Explicit FBC. The distillery is inefficient, because the Explicit FBC suffers of size explosion, but it is a good case study to present distilleries before the optimisations. Moreover, it allows to show an unexpected fact: while adding useful sharing to the calculus will be a quite tricky and technical affair (Sect. XI), adding usefulness to the Open GLAM will be surprisingly simple (Sect. XII), and yet tests of usefulness will only require constant time.

Open GLAM stays for *Open Global LAM*, in turn referring to a similar machine, based on *local* environments, introduced in [15] and called *LAM*—standing for Leroy Abstract Machine. The Open GLAM differs from the LAM in two respects: 1) it uses *global* rather than local environments, and 2) it has an additional rule ($\rightsquigarrow_{\mathtt{c}_3}$) to handle open terms (*i.e.* symbols).

Data-Structures: at the machine level, *terms* are replaced by *codes*, *i.e.* terms not considered up to α-equivalence. To distinguish codes from terms, we over-line codes like in $\overline{t}$.

States (noted $s, s', \ldots$) of the abstract machine are made out of a *context dump* D, a *code* t, an *argument stack* π, and a global environment E, defined by the grammars in Table II. To save space, sometimes we write $[x{\leftarrow}\overline{t}]E$ for $[x{\leftarrow}\overline{t}] : E$. Note that stacks may contain pairs $(\overline{t},\pi)$ of a code and a stack, used to code the application of $\overline{t}$ to the stack π. This representation allows to implement commutative rules in constant time.

The Machine: the machine transitions are given in Table II. Note that the multiplicative one $\rightsquigarrow_{\mathtt{m}}$ puts a new entry in the environment, while the exponential one $\rightsquigarrow_{\mathtt{e}}$ performs a clashing-avoiding substitution from the environment. The idea is that the principal transitions $\rightsquigarrow_{\mathtt{m}}$ and $\rightsquigarrow_{\mathtt{e}}$ implement $\multimap_{\mathtt{m}}$ and $\multimap_{\mathtt{e}}$ while the commutative transitions $\rightsquigarrow_{\mathtt{c}_1}$, $\rightsquigarrow_{\mathtt{c}_2}$, and $\rightsquigarrow_{\mathtt{c}_3}$ locate and expose the next redex following a right-to-left strategy.

The commutative rule $\rightsquigarrow_{\mathtt{c}_1}$ forces evaluation to be right-to-left on applications: the machine processes first the argument $\overline{u}$, saving the left sub term $\overline{t}$ on the dump together with its current stack π. The role of $\rightsquigarrow_{\mathtt{c}_2}$ and $\rightsquigarrow_{\mathtt{c}_3}$ is to backtrack to the saved subterm. Indeed, when the argument, *i.e.* the current code, is finally put in normal form, encoded by a *stack item* ϕ, the stack item is pushed on the stack, and the machine backtracks to the pair on the dump.

The Distillery: machines start an execution on *initial states* defined as $(\epsilon, \overline{t}, \epsilon, \epsilon)$, *i.e.* obtained by taking the term, seen now as the code $\overline{t}$, and setting to ϵ the other machine components. A state represents a term—given by the code—and an evaluation context, that for the Open GLAM is obtained by decoding D, π, and E. The decoding $\underline{\cdot}$ (or distillation) function is defined in Table II. Note that stacks are decoded

Table III
CONTEXT AND RELATIVE UNFOLDING

Context Unfolding			Relative Unfolding			Relative Context Unfolding		
$\langle\cdot\rangle{\downarrow}$	$:=$	$\langle\cdot\rangle$	$t{\downarrow}_{\langle\cdot\rangle}$	$:=$	$t{\downarrow}$	$S'{\downarrow}_{\langle\cdot\rangle}$	$:=$	$S'{\downarrow}$
$(tS){\downarrow}$	$:=$	$t{\downarrow}S{\downarrow}$	$t{\downarrow}_{uS}$	$:=$	$t{\downarrow}_{S}$	$S'{\downarrow}_{uS}$	$:=$	$S'{\downarrow}_{S}$
$(St){\downarrow}$	$:=$	$S{\downarrow}t{\downarrow}$	$t{\downarrow}_{Su}$	$:=$	$t{\downarrow}_{S}$	$S'{\downarrow}_{Su}$	$:=$	$S'{\downarrow}_{S}$
$S[x{\leftarrow}t]{\downarrow}$	$:=$	$S{\downarrow}\{x{\leftarrow}t{\downarrow}\}$	$t{\downarrow}_{S[x\leftarrow u]}$	$:=$	$t{\downarrow}_{S}\{x{\leftarrow}u{\downarrow}\}$	$S'{\downarrow}_{S[x\leftarrow u]}$	$:=$	$S'{\downarrow}_{S}\{x{\leftarrow}u{\downarrow}\}$

to contest in postfix notation for plugging. To improve readability, when we decode machines, we will denote $W\langle t\rangle$ with $\langle t\rangle W$, if the component occurs on the right of t in the machine representation.

A machine state is *closed* when all free variables in any component of the state are bound in E or, equivalently, when $\underline{s}$ is closed in the usual sense. It is *well-named* when all variables bound in the state are distinct. We require well-namedness as a machine invariant to allow every environment entry $[x{\leftarrow}\overline{t}]$ to be global (i.e. to bind x everywhere in the machine state). From now on, the initial state associated to a term t has as code the term obtained α-converting t to make it well-named.

For every machine we will have invariants, in order to prove the properties of a distillery. They are always proved by induction over the length of the execution, by a simple inspection of the transitions. For the Open GLAM:

Lemma 2 (Open GLAM Invariants)**.** *Let $s = (D, u, \pi, E)$ be a state reachable from an initial code $\overline{t}$. Then:*

1) Closure*: s is closed and well-named;*
2) Value*: values in components of s are subterms of $\overline{t}$;*
3) Fireball*: every term in π, in E, and in every stack in D is a fireball;*
4) Contextual Decoding*: $\underline{E}$, $\underline{D}$, $\underline{\pi}$, and F_s are evaluation contexts;*

The invariants are used to prove the following theorem.

Theorem 7 (Open GLAM Distillation)**.** *(Open GLAM, $\multimap_{\mathtt{f}}, \equiv, \underline{\cdot}$) is a reflective explicit distillery. In particular, let s be a reachable state reachable:*

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

Since the Explicit FBC suffers of size explosion, an exponential step (and thus an exponential transition) may duplicate a subterm that is exponentially bigger than the input. Then (Open GLAM, $\multimap_{\mathtt{f}}, \equiv, \underline{\cdot}$) does not satisfy bilinearity, for which every exponential transition has to be linear in the input.

IX. Interlude: Relative Unfoldings

Now we define some notions for weak contexts that will be implicitly instantiated to all kind of contexts in the paper. In particular, we define substitution over contexts, and then use it to define the unfolding of a context, and the more general notion of relative unfolding.

Implicit substitution on weak contexts W is defined by

$$\begin{array}{lcl}
\langle\cdot\rangle\{x{\leftarrow}u\} & := & \langle\cdot\rangle \\
(tW)\{x{\leftarrow}u\} & := & t\{x{\leftarrow}u\}W\{x{\leftarrow}u\} \\
(Wt)\{x{\leftarrow}u\} & := & W\{x{\leftarrow}u\}t\{x{\leftarrow}u\} \\
W[y{\leftarrow}t]\{x{\leftarrow}u\} & := & W\{x{\leftarrow}u\}[y{\leftarrow}t\{x{\leftarrow}u\}] \\
t[y{\leftarrow}W]\{x{\leftarrow}u\} & := & t\{x{\leftarrow}u\}[y{\leftarrow}W\{x{\leftarrow}u\}]
\end{array}$$

Lemma 3. *Let t be a term and W a weak context. Then $W\langle t\rangle\{x{\leftarrow}u\} = W\{x{\leftarrow}u\}\langle t\{x{\leftarrow}u\}\rangle$.*

Now, we would like to extend the unfolding to contexts, but in order to do so we have to restrict the notion of context. Indeed, whenever the hole of a context is inside an ES, the unfolding may erase or duplicate the hole, producing a term or a multi-context, which we do not want. Thus, we turn to (weak) *shallow contexts*, defined by:

$$S, S', S'' ::= \langle\cdot\rangle \mid St \mid tS \mid S[x{\leftarrow}t].$$

(note the absence of the production $t[x{\leftarrow}S]$).

Now, we define in Table III *context unfolding $S{\downarrow}$, unfolding $t{\downarrow}_S$ of a term t relative to a shallow context S* and *unfolding $S'{\downarrow}_S$ of a shallow context S' relative to a shallow context S.*

Relative unfoldings have a number of properties, summed up in the companion technical report [17]. Last, a definition that will be important in the next section.

Definition 4 (Applicative Context)**.** *A shallow context S is* applicative *when its hole is applied to a sub term u, i.e. if $S = S'\langle Lu\rangle$.*

X. Introducing Useful Sharing

Beware: this and the next sections will heavily use contexts and notions about them as defined in Sect. VI and Sect. IX, in particular the notions of *shallow* context, *applicative context*, and *relative unfolding*.

Introducing Useful Reduction: note that the substitution steps in the size exploding family do not create redexes. We want to restrict the calculus so that these *useless* steps are avoided. The idea of useful sharing, is to trigger an exponential redex only if it will somehow contribute to create a multiplicative redex. Essentially, one wants only the exponential steps

$$F\langle x\rangle[x{\leftarrow}L\langle f\rangle] \multimap_{\mathtt{e}} L\langle F\langle f\rangle[x{\leftarrow}f]\rangle$$

s.t. F is applicative and f is a value, so that the firing creates a multiplicative redex. Such a change of approach, however, has consequences on the whole design of the system. Indeed, since some substitutions are delayed, the present requirements for the rules might not be met. Consider:

$$(\lambda x.t)y[y{\leftarrow}ab]$$

we want to avoid substituting ab for the argument y, but we also want that evaluation does not stop, *i.e.* that $(\lambda x.t)y[y{\leftarrow}ab] \rightarrow_{\mathtt{m}} t[x{\leftarrow}y[y{\leftarrow}ab]]$. To accomodate such a dynamics, our definitions have to be *up to unfolding*, *i.e.* *fireballs* have to be replaced by *terms unfolding to fireballs*. There are 4 subtle things about useful reduction.

1) Multiplicatives and Variables: the idea is that the multiplicative rule becomes

$$L\langle\lambda x.t\rangle L'\langle u\rangle \quad \mapsto_{\mathtt{m}} \quad L\langle t[x{\leftarrow}L'\langle u\rangle]\rangle$$

where it is the unfolding $L'\langle u\rangle\!\downarrow$ of the argument $L'\langle u\rangle$ that is a fireball, and not necessarily $L'\langle u\rangle$ itself. Note that sometimes variables are valid arguments of multiplicative redexes, and consequently substitutions may contain variables.

2) Exponentials and Future Creations: the exponential rule involves contexts, and is trickier to make it useful. A first approximation of useful exponential step is

$$F\langle x\rangle[x{\leftarrow}L\langle u\rangle] \quad \mapsto_{\mathtt{e}} \quad L\langle F\langle u\rangle[x{\leftarrow}u]\rangle$$

where $L\langle u\rangle\!\downarrow$ is a *value* (*i.e.* it is not inert) and F is applicative, so that—after eventually many substitution steps, when x becomes $u\!\downarrow$—a multiplicative redex will pop out.

Note that an useful exponential step does not always *immediately* create a multiplicative redex. Consider the following step (where I is the identity):

$$(xI)[x{\leftarrow}y][y{\leftarrow}I] \multimap_{\mathtt{e}} (yI)[x{\leftarrow}y][y{\leftarrow}I] \qquad (1)$$

No multiplicative redex has been created yet, but step (1) is useful because the *next* exponential step creates a multiplicative redex (note how such lookahead is captured by working up to unfoldings):

$$(yI)[x{\leftarrow}y][y{\leftarrow}I] \multimap_{\mathtt{e}} (II)[x{\leftarrow}y][y{\leftarrow}I]$$

3) Evaluation and Evaluable Contexts: the delaying of useless substitutions impacts also on the notion of evaluation context F, used in the exponential rule. For instance, the following exponential step should be useful

$$((xI)y)[x{\leftarrow}I][y{\leftarrow}ab] \multimap_{\mathtt{e}} ((II)y)[x{\leftarrow}I][y{\leftarrow}ab]$$

but the context $((\langle\cdot\rangle I)y)[x{\leftarrow}I][y{\leftarrow}ab]$ isolating x is not an evaluation context, it only unfolds to one. We then need a notion of evaluation context up to unfolding. The intuition is that a shallow context S is *evaluable* if $S\!\downarrow$ is an evaluation context (see Sect. IX for the definition of context unfolding), and it is *useful* if it is evaluable and applicative. The exponential rule then should rather be:

$$S\langle x\rangle[x{\leftarrow}L\langle u\rangle] \quad \mapsto_{\mathtt{e}} \quad L\langle S\langle u\rangle[x{\leftarrow}u]\rangle$$

where $u\!\downarrow$ is a *value* and S is *useful*.

4) Context Closure vs Global Rules: such a definition, while close to the right one, still misses a fundamental point, *i.e.* the *global* nature of useful steps. Evaluation rules are indeed defined by a further *closure by contexts*, *i.e.* a step takes place in a certain shallow context S'. Of course, S' has to be evaluable, but there is more. Such a context, in fact, may also give an essential contribution to the usefulness of a step. Let us give an example. Consider the exponential step

$$(xx)[x{\leftarrow}y] \multimap_{\mathtt{e}} (yx)[x{\leftarrow}y]$$

By itself it is not useful, since y is not a value nor unfolds to one. If we plug that redex in the context $S := \langle\cdot\rangle[y{\leftarrow}I]$, however, then y unfolds to a value in S, as $y\!\downarrow_S = y\!\downarrow_{\langle\cdot\rangle[y\leftarrow\lambda z.z]} = \lambda z.z$, and the step becomes:

$$(xx)[x{\leftarrow}y][y{\leftarrow}\lambda z.z] \multimap_{\mathtt{e}} (yx)[x{\leftarrow}y][y{\leftarrow}\lambda z.z] \qquad (2)$$

As before, no multiplicative redex has been created yet, but step (2) is useful because it is essential for the creation given by the *next* exponential step:

$$(yx)[x{\leftarrow}y][y{\leftarrow}\lambda z.z] \multimap_{\mathtt{e}} ((\lambda z.z)x)[x{\leftarrow}y][y{\leftarrow}\lambda z.z]$$

Note, indeed, that $(\lambda z.z)x$ gives a useful multiplicative redex, because x unfolds to a fireball in its context $\langle\cdot\rangle[x{\leftarrow}y][y{\leftarrow}\lambda z.z]$.

Summing up, the useful or useless character of a step depends crucially on the surrounding context. Therefore useful rules have to be *global*: rather than given as axioms closed by evaluable contexts, they will involve the surrounding context itself and impose conditions about it.

The Useful FBC, presented in the next section, formalises these ideas. We will prove it to be a locally bounded implementation of $\rightarrow_{\mathtt{f}}$, obtaining our fist high-level implementation theorem.

XI. The Useful Fireball Calculus

For the Useful FBC, terms, values, and substitution contexts are unchanged (with respect to the Explicit FBC), and we use *shallow contexts* S as defined in Sect. IX. An *initial term* is still a closed term with no explicit substitutions.

The new key notion is that of *evaluable* context.

Definition 5 (Evaluable and Useful Contexts)**.** Evaluable *(shallow) contexts are defined by the inference system in Table V. A context is* useful *if it is evaluable and applicative (being applicative is easily seen to be preserved by unfolding).*

Point 1 of the following Lemma 4 guarantees that evaluable contexts capture the intended semantics suggested in the previous section. Point 2 instead provides an equivalent inductive formulation that does not mention relative unfoldings. The definition in Table V can be thought has been *from the outside*, while the lemma give a characterisation *from the inside*, relating subterms to their surrounding sub-context.

Lemma 4.

1) *If S is evaluable then $S\!\downarrow$ is an evaluation context.*
2) *S is evaluable iff $u\!\downarrow_{S'}$ is a fireball whenever $S = S'\langle S''u\rangle$ or $S = S'\langle S''[x{\leftarrow}u]\rangle$.*

Table IV
DISTILLERIES IN THE PAPER + REWRITING RULES FOR THE USEFUL FBC

Calculus	*Machine*
FBC $\to_{\mathtt{f}}$	
Explicit FBC $\multimap_{\mathtt{f}}$	Open GLAM
Useful FBC $\multimap_{\mathtt{uf}}$	GLAMOUr
Unchaining FBC $\multimap_{\mathtt{of}}$	Unchaining GLAMOUr

RULE (ALREADY CLOSED BY CONTEXTS)	SIDE CONDITIONS
$S\langle L\langle \lambda x.t\rangle u\rangle \multimap_{\mathtt{um}} S\langle L\langle t[x{\leftarrow}u]\rangle\rangle$	$S\langle Lu\rangle$ is useful
$S\langle S'\langle x\rangle[x{\leftarrow}L\langle u\rangle]\rangle \multimap_{\mathtt{ue}} S\langle L\langle S'\langle u\rangle[x{\leftarrow}u]\rangle\rangle$	$S\langle S'[x{\leftarrow}L\langle u\rangle]\rangle$ is useful $u \neq u'[y{\leftarrow}w]$ and $u{\downarrow}_{S\langle L\rangle} = v$

Table V
EVALUABLE SHALLOW CONTEXTS

$$\frac{}{\langle\cdot\rangle \text{ is evaluable}} \qquad \frac{S \text{ is eval.} \qquad t{\downarrow} \text{ is a fireball}}{St \text{ is evaluable}}$$

$$\frac{S \text{ is evaluable}}{tS \text{ is evaluable}} \qquad \frac{S\{x{\leftarrow}t{\downarrow}\} \text{ is eval.} \qquad t{\downarrow} \text{ is a fireball}}{S[x{\leftarrow}t] \text{ is evaluable}}$$

Rewriting Rules: the two rewriting rules $\multimap_{\mathtt{um}}$ and $\multimap_{\mathtt{ue}}$ are defined in Table IV, and we use $\multimap_{\mathtt{uf}}$ for $\multimap_{\mathtt{um}} \cup \multimap_{\mathtt{ue}}$. The rules are *global*, *i.e.* they do not factor as a rule followed by a contextual closure. As already explained, the context has to be taken into account, to understand if the step is useful to multiplicative redexes.

In rule $\multimap_{\mathtt{um}}$, the requirement that the whole context around the abstraction is useful guarantees that the argument u unfolds to a fireball in its context. Note also that in $\multimap_{\mathtt{ue}}$ this is not enough, as such an unfolding has to be a value, otherwise it will not be useful to multiplicative redexes. Moreover, the rule requires $u \neq u'[y{\leftarrow}w]$, to avoid copying substitutions.

A detailed study of useful evaluation in the companion technical report [17] shows that:

Theorem 8 (Quadratic High-Level Implementation). *$(\to_{\mathtt{f}}, \multimap_{\mathtt{uf}})$ is a locally bounded high-level implementation system, and so it has a quadratic overhead wrt $\to_{\mathtt{f}}$.*

Moreover, the structural equivalence $\equiv$ is a strong bisimulation also with respect to $\multimap_{\mathtt{uf}}$.

Proposition 4 ($\equiv$ is a Strong Bisimulation wrt $\multimap_{\mathtt{uf}}$). *Let $\mathtt{x} \in \{\mathtt{um}, \mathtt{ue}\}$. Then, $t \equiv u$ and $t \multimap_{\mathtt{x}} t'$ implies that there exists u' such that $u \multimap_{\mathtt{x}} u'$ and $t' \equiv u'$.*

XII. THE GLAMOUR MACHINE

Here we refine the Open GLAM with a very simple tagging of stacks and environments, in order to implement useful sharing. The idea is that every term in the stack or in the environment carries a label $l \in \{v, A\}$ indicating if it unfolds (relatively to the environment) to a value or to a inert term.

The grammars are identical to the Open GLAM, up to labels:

$$l ::= v \mid A \qquad E, E' ::= \epsilon \mid [x{\leftarrow}\phi^l] : E$$
$$\pi, \pi' ::= \epsilon \mid \phi^l : \pi$$

The decoding of the various machine components is identical to that for the Open GLAM, up to labels that are ignored. The state context, however, now is noted S_s, as it is not necessarily an evaluation context, but only an evaluable one.

The transitions are in Table VI. They are obtained from those of the Open GLAM by:

1) *Backtracking instead of performing a useless substitution*: there are two new backtracking cases $\rightsquigarrow_{\mathtt{c}_4}$ and $\rightsquigarrow_{\mathtt{c}_5}$ (that in the Open GLAM were handled by the exponential transition), corresponding to avoided useless duplications: $\rightsquigarrow_{\mathtt{c}_4}$ backtracks when the entry ϕ to substitute is marked A (as it unfolds to a inert term) and $\rightsquigarrow_{\mathtt{c}_5}$ backtracks when the term is marked v but the stack is empty (*i.e.* the context is not applicative).
2) *Substituting only when it is useful*: the exponential transition is applied only when the term to substitute has label v and the stack is non-empty.

Lemma 5 (GLAMOUr Invariants). *Let $s = (D, \overline{u}, \pi, E)$ be a state reachable from an initial code $\overline{t}$. Then:*

1) Closure*: s is closed and well named;*
2) Value*: values in components of s are subterms of $\overline{t}$;*
3) Fireball*: $\overline{t}{\downarrow}_{\underline{E}}$ is a fireball (of kind l) for every code $\overline{t}^l$ in π, E, and in every stack of D;*
4) Evaluability*: $\underline{E}$, $\underline{D}{\downarrow}_{\underline{E}}$, $\underline{\pi}{\downarrow}_{\underline{E}}$, and S_s are evaluable contexts;*
5) Environment Size*: the length of the global environment E is bound by $|\rho|_{\mathtt{m}}$.*

Theorem 9 (GLAMOUr Distillation). *$(GLAMOUr, \multimap_{\mathtt{uf}}, \equiv, \underline{\cdot})$ is a reflective explicit distillery. In particular, let s be a reachable state:*

1) Commutative*: if $s \rightsquigarrow_{\mathtt{c}_{1,2,3,4,5}} s'$ then $\underline{s} = \underline{s'}$;*
2) Multiplicative*: if $s \rightsquigarrow_{\mathtt{um}} s'$ then $\underline{s} \multimap_{\mathtt{um}}\equiv \underline{s'}$;*
3) Exponential*: if $s \rightsquigarrow_{\mathtt{ue}} s'$ then $\underline{s} \multimap_{\mathtt{ue}} \underline{s'}$.*

In fact, the distillery is even bilinear, as we now show. The proof employs the following definition of size of a state.

Definition 6. *The* size *of codes and states is defined by:*

$$\begin{array}{rclrcl} |x| = |a| & := & 1 & |\overline{t}\overline{u}| & := & |\overline{t}| + |\overline{u}| + 1 \\ |\lambda x.\overline{t}| & := & |\overline{t}| + 1 & |(D, \overline{t}, \pi, E)| & := & |\overline{t}| + \Sigma_{(\overline{u},\pi) \in D}|\overline{u}| \end{array}$$

Lemma 6 (Size Bounded). *Let $s = (D, \overline{u}, \pi, E)$ be a state reached by an execution ρ of initial code $\overline{t}$. Then $|s| \leq (1 + |\rho|_{\mathtt{ue}})|\overline{t}| - |\rho|_c$.*

Proof: by induction over the length of the derivation. The property trivially holds for the empty derivation. Case analysis over the last machine transition. *Commutative rule* $\rightsquigarrow_{\mathtt{c}_1}$: the rule splits the code $\overline{t}\overline{u}$ between the dump and the code, and the measure—as well as the rhs of the formula—decreases by 1 because the rule consumes the application node. *Commutative rules* $\rightsquigarrow_{\mathtt{c}_{2,3,4,5}}$: these rules consume the current code, so they

Table VI
TRANSITIONS OF THE GLAMOUR

D	$\overline{t}\overline{u}$	π	E	$\rightsquigarrow_{\mathtt{c}_1}$	$D:(\overline{t},\pi)$	$\overline{u}$	ϵ	E
D	$\lambda x.\overline{t}$	$\phi^l:\pi$	E	$\rightsquigarrow_{\mathtt{um}}$	D	$\overline{t}$	π	$[x{\leftarrow}\phi^l]E$
$D:(\overline{t},\pi)$	$\lambda x.\overline{u}$	ϵ	E	$\rightsquigarrow_{\mathtt{c}_2}$	D	$\overline{t}$	$(\lambda x.\overline{u})^v:\pi$	E
$D:(\overline{t},\pi)$	a	π'	E	$\rightsquigarrow_{\mathtt{c}_3}$	D	$\overline{t}$	$(a,\pi')^A:\pi$	E
$D:(\overline{t},\pi)$	x	π'	$E_1[x{\leftarrow}\phi^A]E_2$	$\rightsquigarrow_{\mathtt{c}_4}$	D	$\overline{t}$	$(x,\pi')^A:\pi$	$E_1[x{\leftarrow}\phi^A]E_2$
$D:(\overline{t},\pi)$	x	ϵ	$E_1[x{\leftarrow}\overline{u}^v]E_2$	$\rightsquigarrow_{\mathtt{c}_5}$	D	$\overline{t}$	$x^v:\pi$	$E_1[x{\leftarrow}\overline{u}^v]E_2$
D	x	$\phi^l:\pi$	$E_1[x{\leftarrow}\overline{u}^v]E_2$	$\rightsquigarrow_{\mathtt{ue}}$	D	$\overline{u}^\alpha$	$\phi^l:\pi$	$E_1[x{\leftarrow}\overline{u}^v]E_2$

where $\overline{u}^\alpha$ is any code α-equivalent to $\overline{u}$ that preserves well-naming of the machine.

Table VII
IDENTITY, CHAIN, AND CHAIN-STARTING CONTEXT + REWRITING RULES OF THE UNCHAINING FBC

I, I'	$::=$	$\langle\cdot\rangle \mid I\langle x\rangle[x{\leftarrow}I'] \mid I[x{\leftarrow}t]$
C, C'	$::=$	$S\langle x\rangle[x{\leftarrow}I] \mid C\langle x\rangle[x{\leftarrow}I] \mid S\langle C\rangle$

$\overleftarrow{S\langle y\rangle[y{\leftarrow}I]}^x$	$:=$	$S[y{\leftarrow}I\langle x\rangle]$
$\overleftarrow{C\langle y\rangle[y{\leftarrow}I]}^x$	$:=$	$\overleftarrow{C}^y[y{\leftarrow}I\langle x\rangle]$
$\overleftarrow{S\langle C\rangle}^x$	$:=$	$S\langle\overleftarrow{C}^x\rangle$

RULE (ALREADY CLOSED BY CONTEXTS)	SIDE CONDITION
$S\langle L\langle\lambda x.t\rangle u\rangle \multimap_{\mathtt{om}} S\langle L\langle t[x{\leftarrow}u]\rangle\rangle$	$S\langle\langle\cdot\rangle u\rangle$ is useful
$S\langle S'\langle x\rangle[x{\leftarrow}L\langle v\rangle]\rangle \multimap_{\mathtt{oes}} S\langle L\langle S'\langle v\rangle[x{\leftarrow}v]\rangle\rangle$	$S\langle S'[x{\leftarrow}L\langle v\rangle]\rangle$ is useful
$S\langle C\langle x\rangle[x{\leftarrow}L\langle v\rangle]\rangle \multimap_{\mathtt{oec}} S\langle L\langle C\langle v\rangle[x{\leftarrow}v]\rangle\rangle$	$S\langle\overleftarrow{C}^x[x{\leftarrow}L\langle v\rangle]\rangle$ is useful

decrease the measure of at least 1. *Multiplicative*: it consumes the lambda abstraction. *Exponential*: it modifies the current code by replacing a variable (of size 1) with a value $\overline{v}$ coming from the environment. Because of Lemma 5.2, $\overline{v}$ is a subterm of $\overline{t}$ and the dump size increment is bounded by $|\overline{t}|$. ■

Corollary 1 (Bilinearity of $\rightsquigarrow_{\mathtt{c}}$). *Let s be a state reached by an execution ρ of initial code $\overline{t}$. Then $|\rho|_c \leq (1+|\rho|_{\mathtt{e}})|\overline{t}|$.*

Finally, we obtain our first implementation theorem.

Theorem 10 (Useful Implementation).

1) Low-Level Bilinear Implementation*: a $\multimap_{\mathtt{uf}}$-derivation d is implementable on RAM in $O((1+|d|)\cdot|t|)$ (i.e. bilinear) steps.*
2) Low + High Quadratic Implementation*: a $\rightarrow_{\mathtt{f}}$-derivation d is implementable on RAM in $O((1+|d|^2)\cdot|t|)$ steps, i.e. linear in the size of the initial term t and quadratic in the length of the derivation $|d|$.*

XIII. THE UNCHAINING FBC

In this section we start by analysing why the Useful FBC has a quadratic overhead. We then refine it, obtaining the Unchaining FBC, that we will prove to have only a linear overhead wrt the FBC. The optimisation has to do with the order in which chains of useful substitutions are performed.

Analysis of Useful Substitution Chains: in the Useful FBC, whenever there is a situation like

$$(x_1A)[x_1{\leftarrow}x_2]\ldots[x_{n-1}{\leftarrow}x_n][x_n{\leftarrow}v]$$

the $\multimap_{\mathtt{uf}}$ strategy performs $n+1$ exponential steps $\multimap_{\mathtt{ue}}$ replacing x_1 with x_2, then x_2 with x_3, and so on, until v is finally substituted on the head

$$\begin{array}{l} (x_nA)[x_1{\leftarrow}x_2]\ldots[x_{n-1}{\leftarrow}x_n][x_n{\leftarrow}v] \quad \multimap_{\mathtt{ue}} \\ (vA)[x_1{\leftarrow}x_2]\ldots[x_{n-1}{\leftarrow}x_n][x_n{\leftarrow}v] \end{array}$$

and a multiplicative redex can be fired. Any later occurrence of x_1 will trigger the same chain of exponential steps again. Because the length n of the chain is bounded by the number of previous multiplicative steps (local bound property), the overall complexity of the machine is quadratic in the number of multiplicative steps. In our previous work [16], we showed that to reduce the complexity to linear it is enough to perform substitution steps in reverse order, modifying the chains while traversing them. The idea is that in the previous example one should rather have a smart reduction $\multimap_{\mathtt{oe}}$ ($\mathtt{o}$ stays for optimised, as $\mathtt{u}$ is already used for useful reduction) following the chain of substitutions and performing:

$$\begin{array}{ll} (x_1A)[x_1{\leftarrow}x_2]\ldots[x_{n-1}{\leftarrow}x_n][x_n{\leftarrow}v] & \multimap_{\mathtt{oe}} \\ (x_1A)[x_1{\leftarrow}x_2]\ldots[x_{n-1}{\leftarrow}v][x_n{\leftarrow}v] & \multimap_{\mathtt{oe}} \\ \ldots & \\ (x_1A)[x_1{\leftarrow}v]\ldots[x_{n-1}{\leftarrow}v][x_n{\leftarrow}v] & \multimap_{\mathtt{oe}} \\ (vA)[x_1{\leftarrow}v]\ldots[x_{n-1}{\leftarrow}v][x_n{\leftarrow}v] & \end{array}$$

Later occurrences of x_1 will no longer trigger the chain, because it has been *unchained* by traversing it the first time.

Unfortunately, introducing such an optimisation for useful reduction is hard. In the shown example, that has a very simple form, it is quite easy to define what *following the chain* means. For the distillation machinery to work, however, we need our rewriting rules to be stable by structural equivalence, whose action is a rearrangement of substitutions through the term structure. Then the substitutions $[x_i{\leftarrow}x_{i+1}]$ of the example can be spread all over the term, interleaved by applications and other substitutions, and even nested one into the other (like in $[x_i{\leftarrow}x_{i+1}[x_{i+1}{\leftarrow}x_{i+2}]]$). This makes the specification of *unchaining useful reduction* a quite technical affair.

Chain Contexts: reconsider a term like in the example, $(xA)[x_1{\leftarrow}x_2][x_2{\leftarrow}x_3][x_3{\leftarrow}x_4][x_4{\leftarrow}v]$. We want the next step to substitute on x_4 so we should give a status to the context $C := (xA)[x_1{\leftarrow}x_2][x_2{\leftarrow}x_3][x_3{\leftarrow}\langle\cdot\rangle]$. The problem is that C can be deformed by structural equivalence $\equiv$ as

$$C' := (x[x_1{\leftarrow}x_2[x_2{\leftarrow}x_3]]A)[x_3{\leftarrow}\langle\cdot\rangle]$$

and so this context has to be caught too. We specify these context in Table VII as *chain contexts* C, defined using the auxiliary notion of *identity context* I, that captures a simpler form of chain (note that both notions are not shallow).

Given a chain context C, we will need to retrieve the point where the chain started, *i.e.* the shallow context isolating the variable at the left end of the chain (x_1 in the example). We are now going to define an operation associating to every chain context its *chain-starting (shallow) context*. To see the two as contexts of a same term, we need also to provide the subterm that we will put in C (that will always be a variable). The chain-starting context $\overleftarrow{C}^x$ associated to the chain context C (with respect to x) is defined in Table VII.

For our example $C := (xA)[x_1{\leftarrow}x_2][x_2{\leftarrow}x_3][x_3{\leftarrow}\langle\cdot\rangle]$ we have $\overleftarrow{C}^{x_4} = (\langle\cdot\rangle A)[x_1{\leftarrow}x_2][x_2{\leftarrow}x_3][x_3{\leftarrow}x_4]$, as expected.

Rewriting Rules: the rules of the Unchaining FBC are in Table VII. Note that the exponential rule splits in two, the ordinary *shallow* case $\multimap_{\mathtt{oes}}$ (now constrained to values) and the *chain* case $\multimap_{\mathtt{oec}}$ (where the new definition play a role). They could be merged, but for the complexity analysis and the relationship with the next machine is better to distinguish them. We use $\multimap_{\mathtt{oe}}$ for $\multimap_{\mathtt{oes}} \cup \multimap_{\mathtt{oec}}$, and $\multimap_{\mathtt{of}}$ for $\multimap_{\mathtt{om}} \cup \multimap_{\mathtt{oe}}$. Note the use of $\overleftarrow{C}^x$ in the third side condition.

A. Linearity: Multiplicative vs Exponential Analysis

To prove that $\multimap_{\mathtt{of}}$ implements $\to_{\mathtt{f}}$ with a global bound, and thus with a linear overhead, we need to show that the global number of exponential steps ($\multimap_{\mathtt{oe}}$) in a $\multimap_{\mathtt{of}}$-derivation is bound by the number of multiplicative steps ($\multimap_{\mathtt{om}}$). We need the following invariant.

Lemma 7 (Subterm Invariant)**.** *Let t be a λ-term and $d : t \multimap^* u$. Then every value in u is a value in t.*

A substitution $t[x{\leftarrow}u]$ is *basic* if u has the form $L\langle y\rangle$. The *basic size* $|t|_{\mathtt{b}}$ of t is the number of its basic substitutions.

Lemma 8 (Steps and Basic Size)**.**

1) *If $t \multimap_{\mathtt{oes}} u$ then $|u|_{\mathtt{b}} = |t|_{\mathtt{b}}$;*
2) *If $t \multimap_{\mathtt{oec}} u$ then $|t|_{\mathtt{b}} > 0$ and $|u|_{\mathtt{b}} = |t|_{\mathtt{b}} - 1$;*
3) *If $t \multimap_{\mathtt{om}} u$ then $|u|_{\mathtt{b}} = |t|_{\mathtt{b}}$ or $|u|_{\mathtt{b}} = |t|_{\mathtt{b}} + 1$.*

Lemma 9. *Let t be initial and $d : t \multimap^*_{\mathtt{of}} u$. Then $|u|_{\mathtt{b}} \leq |d|_{\mathtt{om}} - |d|_{\mathtt{oec}}$.*

Proof: by induction on $|d|$. If $|d| = 0$ the statement holds. If $|d| > 0$ consider the last step $w \multimap_{\mathtt{of}} u$ of d and the prefix $e : t \multimap^*_{\mathtt{of}} w$ of d. By *i.h.*, $|w|_{\mathtt{b}} \leq |e|_{\mathtt{om}} - |e|_{\mathtt{oec}}$. Cases of $w \multimap_{\mathtt{of}} u$.

Shallow Exponential Step $\multimap_{\mathtt{oes}}$:

$$\begin{array}{lll} |u|_{\mathtt{b}} & \leq_{L.8.1} & |w|_{\mathtt{b}} - 1 \\ & \leq_{i.h.} & |e|_{\mathtt{om}} - |e|_{\mathtt{oec}} - 1 \\ & = & |e|_{\mathtt{om}} - (|e|_{\mathtt{oec}} + 1) \quad = \quad |d|_{\mathtt{om}} - |d|_{\mathtt{oec}} \end{array}$$

Chain Exponential Step $\multimap_{\mathtt{oec}}$:

$$|u|_{\mathtt{b}} =_{L.8.2} |w|_{\mathtt{b}} \leq_{i.h.} |e|_{\mathtt{om}} - |e|_{\mathtt{oec}} = |d|_{\mathtt{om}} - |d|_{\mathtt{oec}}$$

Multiplicative Step $\multimap_{\mathtt{om}}$:

$$\begin{array}{lll} |u|_{\mathtt{b}} & \leq_{L.8.3} & |w|_{\mathtt{b}} + 1 \\ & \leq_{i.h.} & |e|_{\mathtt{om}} - |e|_{\mathtt{oec}} + 1 \\ & = & e + 1 - |e|_{\mathtt{oec}} \quad = \quad |d|_{\mathtt{om}} - |d|_{\mathtt{oec}} \end{array}$$ ■

Corollary 2 (Linear Bound on Chain Exponential Steps)**.** *Let t be initial and $d : t \multimap^*_{\mathtt{of}} u$. Then $|d|_{\mathtt{oec}} \leq |d|_{\mathtt{om}}$.*

Next, we bound shallow steps.

Lemma 10 (Linear Bound on Shallow Exponential Steps)**.** *Let t be initial and $d : t \multimap^*_{\mathtt{of}} u$. Then $|d|_{\mathtt{oes}} \leq |d|_{\mathtt{om}}$.*

Proof: first note that if $t \multimap_{\mathtt{oes}} u$ then $u \multimap_{\mathtt{om}} w$, because by definition $\multimap_{\mathtt{oes}}$ can fire only if it creates a $\multimap_{\mathtt{om}}$-redex. Such a fact and determinism of $\multimap_{\mathtt{of}}$ together imply $|d|_{\mathtt{oes}} \leq |d|_{\mathtt{om}} + 1$, because every $\multimap_{\mathtt{oes}}$ step is matched by the eventual $\multimap_{\mathtt{om}}$ steps that follows it immediately. However, note that in t there are no explicit substitutions so that the first step is necessarily an unmatched $\multimap_{\mathtt{om}}$ step. Thus $|d|_{\mathtt{oes}} \leq |d|_{\mathtt{om}}$. ■

Theorem 11 (Linear Bound on Exponential Steps)**.** *Let t be initial and $d : t \multimap^*_{\mathtt{of}} u$. Then $|d|_{\mathtt{oe}} \leq 2 \cdot |d|_{\mathtt{om}}$.*

Proof: by definition, $|d|_{\mathtt{oe}} = |d|_{\mathtt{oec}} + |d|_{\mathtt{oes}}$. By Corollary 2, $|d|_{\mathtt{oec}} \leq |d|_{\mathtt{om}}$ and by Lemma 10 $|d|_{\mathtt{oes}} \leq |d|_{\mathtt{om}}$, and so $|d|_{\mathtt{oe}} \leq 2 \cdot |d|_{\mathtt{om}}$. ■

We presented the interesting bit of the proof of our improved high-level implementation theorem, which follows. The remaining details are in [17].

Theorem 12 (Linear High-Level Implementation)**.** *$(\to_{\mathtt{f}}, \multimap_{\mathtt{of}})$ is a globally bounded high-level implementation system, and so it has a linear overhead wrt $\to_{\mathtt{f}}$.*

Last, the structural equivalence $\equiv$ is a strong bisimulation also for the Unchaining FBC.

Proposition 5 ($\equiv$ is a Strong Bisimulation)**.** *Let $\mathtt{x} \in \{\mathtt{om}, \mathtt{oms}, \mathtt{omc}\}$. Then, $t \equiv u$ and $t \multimap_{\mathtt{x}} t'$ implies that there exists u' such that $u \multimap_{\mathtt{x}} u'$ and $t' \equiv u'$.*

XIV. Unchaining GLAMOUr

The Unchaining GLAMOUr machine, in Table VIII, behaves like the GLAMOUr machine until the code is a variable x_1 that is hereditarily bound in the global environment to a value via the chain $[x_1{\leftarrow}x_2]^v \ldots [x_n{\leftarrow}v]^v$, and the stack is not empty (*i.e.* evaluation is in an applicative context). At this point the machine needs to traverse the chain until it finds the final binding $[x_n{\leftarrow}v]^v$, and then traverse again the chain in the opposite direction replacing every $[x_i{\leftarrow}x_{i+1}]^v$ entry with $[x_i{\leftarrow}v]^v$.

The forward traversal of the chain is implemented by a new commutative rule $\rightsquigarrow_{\mathtt{c}_6}$ that pushes the variables encountered in the chain on a new machine component, called the *chain heap*. The backward traversal is driven by the next variable popped from the heap, and it is implemented by a new exponential rule (the *chain* exponential rule, corresponding to that of the calculus). Most of the analyses of the GLAMOUr carry over to the Unchaining GLAMOUr.

Table VIII
TRANSITIONS OF THE UNCHAINING GLAMOUR

D	ϵ	$\overline{t}\overline{u}$	π	E	$\rightsquigarrow_{\mathtt{c}_1}$	$D:(\overline{t},\pi)$	ϵ	$\overline{u}$	ϵ	E
D	ϵ	$\lambda x.\overline{t}$	$\phi^l:\pi$	E	$\rightsquigarrow_{\mathtt{om}}$	D	ϵ	$\overline{t}$	π	$[x{\leftarrow}\phi^l]E$
$D:(\overline{t},\pi)$	ϵ	$\lambda x.\overline{u}$	ϵ	E	$\rightsquigarrow_{\mathtt{c}_2}$	D	ϵ	$\overline{t}$	$(\lambda x.\overline{u})^v:\pi$	E
$D:(\overline{t},\pi)$	ϵ	a	π'	E	$\rightsquigarrow_{\mathtt{c}_3}$	D	ϵ	$\overline{t}$	$(a,\pi')^A:\pi$	E
$D:(\overline{t},\pi)$	ϵ	x	π'	$E_1[x{\leftarrow}\phi^A]E_2$	$\rightsquigarrow_{\mathtt{c}_4}$	D	ϵ	$\overline{t}$	$(x,\pi')^A:\pi$	$E_1[x{\leftarrow}\phi^A]E_2$
$D:(\overline{t},\pi)$	ϵ	x	ϵ	$E_1[x{\leftarrow}\overline{u}^v]E_2$	$\rightsquigarrow_{\mathtt{c}_5}$	D	ϵ	$\overline{t}$	$x^v:\pi$	$E_1[x{\leftarrow}\overline{u}^v]E_2$
D	ϵ	x	$\phi^l:\pi$	$E_1[x{\leftarrow}\overline{v}^v]E_2$	$\rightsquigarrow_{\mathtt{oes}}$	D	ϵ	$\overline{v}^\alpha$	$\phi^l:\pi$	$E_1[x{\leftarrow}\overline{v}^v]E_2$
D	H	x	$\phi^l:\pi$	$E_1[x{\leftarrow}y^v]E_2$	$\rightsquigarrow_{\mathtt{c}_6}$	D	$H:x$	y	$\phi^l:\pi$	$E_1[x{\leftarrow}y^v]E_2$
D	$H:y$	x	$\phi^l:\pi$	$E^\bullet$	$\rightsquigarrow_{\mathtt{oec}}$	D	H	y	$\phi^l:\pi$	E°

with $E^\bullet := E_1[x{\leftarrow}\overline{v}^v]E_2[y{\leftarrow}x^v]E_3$, $E^\circ := E_1[x{\leftarrow}\overline{v}^v]E_2[y{\leftarrow}\overline{v}^{\alpha\,v}]E_3$, and where $\overline{v}^\alpha$ is any code α-equivalent to $\overline{v}$ that preserves well-naming of the machine.

Every old grammar is as before, and heaps are simply lists of variables, *i.e.* they are defined by $H ::= \epsilon \mid H : x$.

Decoding and Invariants: because of chain heaps and chain contexts, the decoding is involved.

First of all, note that there is a correlation between the chain and the environment, as the variables of a chain heap $H = x_1 : \ldots : x_n$ need to have corresponding entries $[x_i{\leftarrow}x^v_{i+1}]$. More precisely, we will show that the following notion of compatibility is an invariant of the machine.

Definition 7 (Compatibility Heap-Environment)**.** *Let E be an environment and $H = x_1 : \ldots : x_n$ be a heap. We say that H is* compatible *with E if either H is empty or $[x_i{\leftarrow}x^v_{i+1}] \in E$ for $i < n$, $[x_n{\leftarrow}x^v] \in E$, and $[x{\leftarrow}\phi^v] \in E$ for some ϕ^v.*

Given a state $s = (D, H, \overline{t}, \pi, E)$, the dump, the stack and the environment provide a shallow context $S_s := \langle \underline{D}\langle\underline{\pi}\rangle\rangle\underline{E}$ that will be shown to be evaluable, as for the GLAMOUr.

If the chain heap H is not empty, the current code $\overline{t}$ is somewhere in the middle of a chain inside the environment, and it is not apt to fill the state context S_s. The right code is the variable x_1 starting the chain heap $H = x_1 : \ldots : x_n$. Thus, the term to plug in the state context is $\overline{t}^H$, given by:

$$\overline{t}^\epsilon \quad := \quad \overline{t} \qquad\qquad \overline{t}^{x_1:\ldots:x_n} \quad := \quad x_1$$

Finally, a state decodes to a term as follows: $\underline{s} := S_s\langle\overline{t}^H\rangle$.

Lemma 11 (Unchaining GLAMOUr Invariants)**.** *Let $s = (D, H, \overline{u}, \pi, E)$ be a state reachable from an initial code $\overline{t}$.*

1) Closure*: s is closed and s is well named;*
2) Value*: values in components of s are subterms of $\overline{t}$;*
3) Fireball*: $\overline{t}{\downarrow}_{\underline{E}}$ is a fireball (of kind l) for every code $\overline{t}^l$ in π and E;*
4) Evaluability*: $\underline{E}$, $\underline{D}{\downarrow}_{\underline{E}}$, $\underline{\pi}{\downarrow}_{\underline{E}}$, and S_s are evaluable cont.;*
5) Environment Size*: the length of the global environment E is bound by $|\rho|_{\mathtt{m}}$.*
6) Compatible Heap*: if $H \neq \epsilon$ then the stack is not empty, $\overline{u} = x$, and H is compatible with E.*

We need additional decodings to retrieve the chain-starting context C in the side-condition of $\multimap_{\mathtt{oec}}$ rule, that—unsurprisingly—is given by the chain heap. Let $s = (D, H : y, \overline{t}, \pi, E)$ be a state s.t. $H : y$ is compatible with E. Note that compatibility gives $E = E_1[y{\leftarrow}\overline{t}^v]E_2$. Define the chain context C_s and the substitution context L_s as:

$$C_s \quad := \quad \langle\underline{D}\langle\langle y^H\rangle\underline{\pi}\rangle\rangle\underline{E_1}[y{\leftarrow}\langle\cdot\rangle] \qquad\qquad L_s \quad := \quad \underline{E_2}$$

The first point of the following lemma guarantees that C_s and L_s are well defined. The second point proves that filling $L_s\langle C_s\rangle$ with the current term gives exactly the decoding of the state $\underline{s} = S_s\langle y^H\rangle$, and moreover the chain starts exactly on the evaluable context given by the state, *i.e.* that $S_s = L_s\langle\overleftarrow{C_s}^x\rangle$.

Lemma 12 (Heaps and Contexts)**.** *Let $s = (D, H : y, x, \pi, E)$ be a state s.t. $H : y$ is compatible with E. Then:*

1) *L_s is a substitution context and C_s is a chain context*
2) *s. t. $\underline{s} = S_s\langle y^H\rangle = L_s\langle C_s\langle x\rangle\rangle$ with $S_s = L_s\langle\overleftarrow{C_s}^x\rangle$*

We can now sum up.

Theorem 13 (Unchaining GLAMOUr Distillation)**.** *(Unchaining GLAMOUr, $\multimap_{\mathtt{of}}, \equiv, \underline{\cdot}$) is a reflective explicit distillery. In particular, let s be a reachable state:*

1) Commutative*: if $s \rightsquigarrow_{\mathtt{c}_{1,2,3,4,5,6}} s'$ then $\underline{s} = \underline{s'}$;*
2) Multiplicative*: if $s \rightsquigarrow_{\mathtt{om}} s'$ then $\underline{s} \multimap_{\mathtt{om}}\equiv \underline{s'}$;*
3) Shallow Exponential*: if $s \rightsquigarrow_{\mathtt{oes}} s'$ then $\underline{s} \multimap_{\mathtt{oes}} \underline{s'}$;*
4) Chain Exponential*: if $s \rightsquigarrow_{\mathtt{oec}} s'$ then $\underline{s} \multimap_{\mathtt{oec}} \underline{s'}$.*

A. Bilinearity: Principal vs Commutative Analysis

Bilinearity wrt $\rightsquigarrow_{\mathtt{c}_{1,2,3,4,5}}$ is identical to that of the GLAMOUr, thus we omit it and focus on $\rightsquigarrow_{\mathtt{c}_6}$.

The size $|H|$ of a chain heap is its length as a list.

Lemma 13 (Linearity of $\rightsquigarrow_{\mathtt{c}_6}$)**.** *Let $s = (D, H, \overline{t}, \pi, E)$ be a state reached by an execution ρ. Then*

1) $|\rho|_{\mathtt{c}_6} = |H| + |\rho|_{\mathtt{oec}}$.
2) $|H| \leq |\rho|_{\mathtt{m}}$.
3) $|\rho|_{\mathtt{c}_6} \leq |\rho|_{\mathtt{m}} + |\rho|_{\mathtt{oec}} = O(|\rho|_p)$.

Proof: 1) By induction over $|\rho|$ and analysis of the last machine transition. The $\rightsquigarrow_{\mathtt{c}_6}$ steps increment the size of the heap. The $\rightsquigarrow_{\mathtt{oec}}$ steps decrement it. All other steps do not change the heap. *2)* By the compatible heap invariant (Lemma 11.6), $|H| \leq |E|$. By the environment size invariant (Lemma 11.5), $|E| \leq |\rho|_{\mathtt{m}}$. Then $|H| \leq |\rho|_{\mathtt{m}}$. *3)* Plugging Point 2 into Point 1. ■

Corollary 3 (Bilinearity of $\rightsquigarrow_{\mathtt{c}}$)**.** *Let s be a state reached by an execution ρ of initial code $\overline{t}$. Then $|\rho|_{\mathtt{c}} \leq (1 + |\rho|_{\mathtt{e}})|\overline{t}| + |\rho|_{\mathtt{m}} + |\rho|_{\mathtt{oec}} = O((1 + |\rho|_p) \cdot |\overline{t}|)$.*

Finally, we obtain the main result of the paper.

Theorem 14 (Useful Implementation)**.**

1) Low-Level Bilinear Implementation*: a* $\multimap_{\mathtt{of}}$*-derivation* d *is implementable on RAM in* $O((1 + |d|) \cdot |t|)$ *steps.*
2) Low + High Bilinear Implementation*: a* $\to_{\mathtt{f}}$*-derivation* d *is implementable on RAM in* $O((1 + |d|) \cdot |t|)$ *steps.*

Let us conclude with a remark. Our result requires a compact representation of terms via ES. Because unfolding may exponentially increase the size of a term, it is important to show that common operations like equality checking (up to α-conversion) can be implemented efficiently on the compact representation. In other words, ES are *succinct data structures*, in the sense of Jacobson [37].

Despite quadratic and quasi-linear recent algorithms [6], [38] for testing equality of terms with ES, we discovered that a linear algorithm can be obtained by slightly modifying an algorithm already known quite some time before (1976!): the Paterson-Wegman linear unification algorithm [39] (better explained in [40]). The algorithm works on first order terms represented as DAGs, and unification boils down to equality checking when no metavariable occurs in the involved terms.

Our terms with ES can not be fed directly to the Paterson-Wegmar algorithm: we represent shared terms via occurrences of variables bound in substitutions, whereas Paterson-Wegmar uses a simple DAG representation. The change of representation can be easily done in linear time in the size of the input.

Moreover, the Paterson-Wegmar algorithm works with standard equality, whereas we are interested in α-equivalence. Therefore the algorithm needs to be adapted so that two λ-bound variables are considered equivalent when they point to binding nodes that have already been determined to be candidates for equality. The details of the adaptation of Paterson-Wegmar are left to a forthcoming publication.

Acknowledgements

A special acknowledgement to Ugo Dal Lago, to whom we owe the intuition that using labels may lead to a local and efficient implementation of useful sharing. We are also grateful to François Pottier, whose comments on a draft helped to improve the terminology and the presentation.

References

[1] B. Accattoli and U. Dal Lago, "Beta Reduction is Invariant, Indeed," in *CSL-LICS 2014*, 2014, p. 8.

[2] G. E. Blelloch and J. Greiner, "Parallelism in sequential functional languages," in *FPCA*, 1995, pp. 226–237.

[3] D. Sands, J. Gustavsson, and A. Moran, "Lambda calculi and linear speedups," in *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, 2002, pp. 60–84.

[4] U. Dal Lago and S. Martini, "The weak lambda calculus as a reasonable machine," *Theor. Comput. Sci.*, vol. 398, no. 1-3, pp. 32–50, 2008.

[5] ——, "On constructor rewrite systems and the lambda calculus," *Logical Methods in Computer Science*, vol. 8, no. 3, 2012.

[6] B. Accattoli and U. Dal Lago, "On the invariance of the unitary cost model for head reduction," in *RTA*, 2012, pp. 22–37.

[7] B. Accattoli, E. Bonelli, D. Kesner, and C. Lombardi, "A nonstandard standardization theorem," in *POPL*, 2014, pp. 659–670.

[8] B. Grégoire and X. Leroy, "A compiled implementation of strong reduction," in *(ICFP '02)*, 2002, pp. 235–246.

[9] L. Paolini and S. Ronchi Della Rocca, "Call-by-value solvability," *ITA*, vol. 33, no. 6, pp. 507–534, 1999.

[10] B. Accattoli and L. Paolini, "Call-by-value solvability, revisited," in *FLOPS*, 2012, pp. 4–16.

[11] A. Carraro and G. Guerrieri, "A semantical and operational account of call-by-value solvability," in *FOSSACS 2014*, 2014, pp. 103–118.

[12] J.-J. Lévy, "Réductions correctes et optimales dans le lambda-calcul," Thése d'Etat, Univ. Paris VII, France, 1978.

[13] R. Milner, M. Tofte, R. Harper, and D. Macqueen, *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

[14] D. Clément, T. Despeyroux, G. Kahn, and J. Despeyroux, "A simple applicative language: Mini-ml," in *LFP '86*. New York, NY, USA: ACM, 1986, pp. 13–27.

[15] B. Accattoli, P. Barenbaum, and D. Mazza, "Distilling abstract machines," in *ICFP 2014*, 2014, pp. 363–376.

[16] B. Accattoli and C. Sacerdoti Coen, "On the value of variables," in *WoLLIC 2014*, 2014, pp. 36–50.

[17] ——, "On the Relative Usefullness of Fireballs," arXiv:1505.03791, pp. 1–34, 2015, pre-print with Technical Appendix. [Online]. Available: http://arxiv.org/abs/1505.03791

[18] R. Milner, "Local bigraphs and confluence: Two conjectures," *Electr. Notes Theor. Comput. Sci.*, vol. 175, no. 3, pp. 65–73, 2007.

[19] D. Kesner and S. Ó. Conchúir, "Milner's lambda calculus with partial substitutions," Paris 7 University, Tech. Rep., 2008.

[20] B. Accattoli and D. Kesner, "The structural λ-calculus," in *CSL*, 2010, pp. 381–395.

[21] N. G. de Bruijn, "Generalizing Automath by Means of a Lambda-Typed Lambda Calculus," in *Mathematical Logic and Theoretical Computer Science*, ser. Lecture Notes in Pure and Applied Mathematics, no. 106. Marcel Dekker, 1987, pp. 71–92.

[22] R. P. Nederpelt, "The fine-structure of lambda calculus," Eindhoven Univ. of Technology, Tech. Rep. CSN 92/07, 1992.

[23] B. Accattoli, "An abstract factorization theorem for explicit substitutions," in *RTA*, 2012, pp. 6–21.

[24] P. Curien, "An abstract framework for environment machines," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 389–402, 1991.

[25] T. Hardin and L. Maranget, "Functional runtime systems within the lambda-sigma calculus," *J. Funct. Program.*, vol. 8, no. 2, pp. 131–176, 1998.

[26] M. Biernacka and O. Danvy, "A concrete framework for environment machines," *ACM Trans. Comput. Log.*, vol. 9, no. 1, 2007.

[27] F. Lang, "Explaining the lazy Krivine machine using explicit substitution and addresses," *Higher-Order and Symbolic Computation*, vol. 20, no. 3, pp. 257–270, 2007.

[28] P. Crégut, "Strongly reducing variants of the Krivine abstract machine," *Higher-Order and Symbolic Computation*, vol. 20, no. 3, pp. 209–230, 2007.

[29] Z. M. Ariola, A. Bohannon, and A. Sabry, "Sequent calculi and abstract machines," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 4, 2009.

[30] M. Fernández and N. Siafakas, "New developments in environment machines," *Electr. Notes Theor. Comput. Sci.*, vol. 237, pp. 57–73, 2009.

[31] O. Danvy and I. Zerny, "A synthetic operational account of call-by-need evaluation," in *PPDP*, 2013, pp. 97–108.

[32] V. Danos and L. Regnier, "Head linear reduction," Tech. Rep., 2004.

[33] O. Danvy and L. R. Nielsen, "Refocusing in reduction semantics," BRICS, Tech. Rep. RS-04-26, 2004.

[34] M. Wand, "On the correctness of the krivine machine," *Higher-Order and Symbolic Computation*, vol. 20, no. 3, pp. 231–235, 2007.

[35] D. P. Friedman, A. Ghuloum, J. G. Siek, and O. L. Winebarger, "Improving the lazy krivine machine," *Higher-Order and Symbolic Computation*, vol. 20, no. 3, pp. 271–293, 2007.

[36] P. Sestoft, "Deriving a lazy abstract machine," *J. Funct. Program.*, vol. 7, no. 3, pp. 231–264, 1997.

[37] G. J. Jacobson, "Succinct static data structures," Ph.D. dissertation, Pittsburgh, PA, USA, 1988, aAI8918056.

[38] C. Grabmayer and J. Rochel, "Maximal sharing in the lambda calculus with letrec," in *ICFP 2014*, 2014, pp. 67–80.

[39] M. S. Paterson and M. N. Wegman, "Linear unification," in *STOC '76*. New York, NY, USA: ACM, 1976, pp. 181–186.

[40] D. de Champeaux, "About the Paterson-Wegman linear unification algorithm," *J. Comput. Syst. Sci.*, vol. 32, no. 1, pp. 79–90, Feb. 1986.