Crumbling Abstract Machines

Beniamino Accattoli, Andrea Condoluci, Giulio Guerrieri, Claudio Coen

Crumbling Abstract Machines

Beniamino Accattoli
LIX
Inria & École Polytechnique
Palaiseau, France
beniamino.accattoli@inria.fr

Andrea Condoluci
Department of Computer Science and Engineering
University of Bologna
Bologna, Italy
andrea.condoluci@unibo.it

Giulio Guerrieri
Department of Computer Science
University of Bath
Bath, United Kingdom
g.guerrieri@bath.ac.uk

Claudio Sacerdoti Coen
Department of Computer Science and Engineering
University of Bologna
Bologna, Italy
claudio.sacerdoticoen@unibo.it

ABSTRACT

Extending the λ-calculus with a construct for sharing, such as let expressions, enables a special representation of terms: iterated applications are decomposed by introducing sharing points in between any two of them, reducing to the case where applications have only values as immediate subterms.

This work studies how such a crumbled representation of terms impacts on the design and the efficiency of abstract machines for call-by-value evaluation. About the design, it removes the need for data structures encoding the evaluation context, such as the applicative stack and the dump, that get encoded in the environment. About efficiency, we show that there is no slowdown, clarifying in particular a point raised by Kennedy, about the potential inefficiency of such a representation.

Moreover, we prove that everything smoothly scales up to the delicate case of open terms, needed to implement proof assistants. Along the way, we also point out that continuation-passing style transformations—that may be alternatives to our representation—do not scale up to the open case.

KEYWORDS

abstract machine, complexity, explicit substitution, lambda-calculus

ACM Reference Format:
Beniamino Accattoli, Andrea Condoluci, Giulio Guerrieri, and Claudio Sacerdoti Coen. 2019. Crumbling Abstract Machines. In *Principles and Practice of Programming Languages 2019 (PPDP '19), October 7–9, 2019, Porto, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3354166.3354169

1 INTRODUCTION

This paper is about the extension of λ-calculus with explicit constructors for *sharing*. The simplest such construct is a let $x = u$ in t expression, standing for t *where* x *will be substituted by* u, that we also write more concisely as $t[x\leftarrow u]$ and call ES (for *explicit sharing*, or *explicit subsitution*[1]). Thanks to ES, β-reduction can be decomposed into more atomic steps. The simplest decomposition splits β-reduction as $(\lambda x.t)u \to_{\beta_{\mathrm{ES}}} t[x\leftarrow u] \to_{\mathrm{ES}} t\{x\leftarrow u\}$ where $t\{x\leftarrow u\}$ is the meta-level substitution of u for the free occurrences of x in t.

It is well-known that ES are somewhat redundant, as they can always be removed, by simply coding them as β-redexes. They are however more than syntactic sugar, as they provide a simple and yet remarkably effective tool to understand, implement, and program with λ-calculi and functional programming languages.

From a logical point of view, ES are the proof terms corresponding to the extension of natural deduction with a cut rule, and the cut rule is *the* rule representing computation, according to Curry-Howard. From an operational semantics point of view, they allow elegant formulations of subtle strategies such as call-by-need evaluation—various presentations of call-by-need use ES [13, 25, 27, 28, 34, 35] and a particularly simple one is in Accattoli et al. [3]. From a programming point of view, they are part of most functional languages we are aware of. From a rewriting point of view, they enable proof techniques not available within the λ-calculus (*e.g.* reducing a global rewriting properties such as standardization to a local form, see Accattoli [1]). Finally, sharing is used in all implementations of tools based on the λ-calculus to circumvent *size explosion*, the degenerate behavior for which the size of λ-terms may grow exponentially with the number of β-steps.

Crumbled forms. Once sharing is added to the λ-calculus, it enables a representation of terms where a sharing point is associated with every constructor of the term. Such a special form, roughly, is obtained by (recursively) decomposing iterated applications by introducing an ES in between any two of them. For instance, the representation of the term $(((\lambda x.x(xx))y)((\lambda z.z)y))y$ is

$$(w''y)[w''\leftarrow w'w][w'\leftarrow(\lambda x.(xx')[x'\leftarrow xx])y][w\leftarrow(\lambda z.z)y]$$

Note that the transformation involves also function bodies (*i.e.* $\lambda x.x(xx)$ turns into $\lambda x.(xx')[x'\leftarrow xx]$), that ES are grouped together

PPDP '19, October 7–9, 2019, Porto, Portugal

ACM ISBN 978-1-4503-7249-7/19/10…$15.00
https://doi.org/10.1145/3354166.3354169

[1] let expressions and explicit substitutions usually come with different operational semantics: let expressions substitute in just one step, while explicit substitutions substitute in many *micro* steps, percolating through the term structure. They follow however the same typing principles. Moreover, explicit substitutions have many different formulations. In this paper we see let expressions as yet another form of explicit substitutions, and thus conflate the two terminologies.

unless forbidden by abstractions, and that ES are flattened out, i.e. they are not nested unless nesting is forced by abstractions.

This work studies such a representation, called *crumbled* as it crumbles a term by means of ES. Our *crumbling translation* closely resembles—while not being exactly the same—the transformation into *a(dministrative) normal form* (shortened to ANF) introduced by Flanagan et al. [18], building on work by Sabry and Felleisen [33], itself a variant of the *continuation-passing style* (CPS) translation.

A delicate point is to preserve crumbled forms during evaluation. ES often come together with commutation rules to move them around the term structure. These rules are often used to unveil redexes during evaluation or to preserve specific syntactic forms. They may introduce significant overhead that, if not handled carefully, can even lead to asymptotic slowdowns as shown by Kennedy [24]. One of the contributions of this work is to show that crumbled forms can be evaluated and preserved with no need of commutation rules, therefore avoiding Kennedy's potential slowdown.

This paper. Our focus is on the impact of crumbled forms on the design and asymptotic overhead of abstract machines with *weak* evaluation (*i.e.* out of abstractions) on closed terms, and the scalability to (possibly) open terms. Bounding the overhead of abstract machines is a new trend, according to which the machine overhead has to be proved polynomial or even linear in the number of β-steps [2–5, 9, 11]. Open terms—that are not needed to implement functional languages—are used to implement the more general and subtle case of proof assistants. The two topics actually motivate each other: the naive handling of open terms with the techniques for functional languages gives abstract machines with exponential overhead [9, 11], which pushes to develop more efficient machines.

We anticipate here the main results of the paper: crumbled forms induce abstract machines for weak evaluation with less data structures and the transformation does not introduce any asymptotic overhead. Moreover, these facts smoothly scale up to open terms.

Why study crumbled forms. Our interest in studying crumbled forms comes precisely from the fact that they remove some data structures from the design of abstract machines. The relevance of this fact becomes evident when one tries to design abstract machines for strong evaluation (that is, evaluating under abstraction). The study of such machines is extremely technical (see also section Sect. 8) because they have more data structures and more transitions than in the closed and open cases. The many additional transitions are in particular due to the handling of the various data structures. In call-by-name, the situation is still manageable [2, 4, 14, 19], but in call-by-value/need the situation becomes quickly desperate—it is not by chance that there is not a single strong abstract machine for call-by-value/need in the literature.

This work is then preliminary to a detailed study of strong abstract machines for call-by-value and call-by-need. The aim is to explore the subtleties in frameworks that are well understood, such as the closed and open call-by-value cases, and show that there are no slowdowns in turning to a crumbled representation.

The next sub-sections continue the introduction with a lengthy overview of the role of environments, the content of the paper, the relationship with the ANF, the asymptotic study of abstract machines, and related work.

1.1 Environments

ES are often grouped together instead of being scattered all over the term, in finite sequences called *environments*. Abstract machines typically rely on environments. Crumbled forms also rely on packing ES together, as pointed out before, but depart from the ordinary case as environments may appear also under abstractions.

Crumbled Environments. The notion of environment induced by crumbled forms, named here *crumbled environments*, is peculiar. Crumbled environments indeed play a double role: they both store delayed substitutions, as also do ordinary environments, and *encode evaluation contexts*. In ordinary abstract machines, the evaluation context is usually stored in data structures such as the *applicative stack* or the *dump*. Roughly, they implement the search for the redex in the ordinary applicative structure of terms. For crumbled forms, the evaluation context is encoded in the crumbled environment, and so the other structures disappear.

Operations on Crumbled Environments. There are two subtle implementative aspects of crumbled environments, that set them apart from ordinary ones. Ordinary environments are presented with a sequential structure but they are only accessed randomly (that is, not sequentially)—in other words, their sequential structure does not play a role. Crumbled environments, as the ordinary ones, are accessed randomly, to retrieve delayed substitutions, but they are also explored sequentially—since they encode evaluation contexts—in order to search for redexes. Therefore, their implementation has to reflect the sequential structure.

The second subtlety is that crumbled machines also have to concatenate environments, that is an operation never performed by ordinary machines, and that has to be concretely implemented as efficiently as possible, *i.e.* in constant time. That this point is subtle is proved by the fact that Kennedy's slowdown [24] amounts to a quadratic overhead in evaluating terms in ANF due to the concatenation of environments.

To address these points, we provide a prototype OCaml implementation of crumbled environments in in the appendix of [6], to be compared with the one of global environments in Accattoli and Barras [5], that does not concretely implement the sequential structure. In particular, our implementation concatenates environments in constant time and does not suffer from Kennedy's slowdown. Essentially, Kennedy's slowdown amounts to the fact that his implementation concatenates ANF environments in linear rather than constant time (see Section 9).

1.2 Content of the Paper

The Closed Case. First, we define crumbled forms and an abstract machine evaluating them, the Crumble GLAM, and show that it implements Plotkin's *closed* small-step call-by-value (CbV for short) λ-calculus (extended with conditionals, see below). Moreover, we study the overhead of the machine, and show that it is linear in the number of β-steps and in the size of the initial term, exactly as the best machines for CbV executing ordinary terms. Therefore, the crumbling transformation does not introduce any asymptotic overhead. The study is detailed and based on a careful and delicate spelling of the invariants of the machine. In particular, our approach does not suffer from Kennedy's potential slowdown.

Open Terms. The second ingredient of the new trend of abstract machines [2–5, 9, 11]—the first being complexity analyses—is studying evaluation in presence of (possibly) open terms or even strong evaluation (*i.e.* under abstraction), which is required in order to implement proof assistants. Apart from few exceptions—Crégut [14], Grégoire and Leroy [20], and García-Pérez et al. [19]—the literature before the new wave mostly neglected these subtle cases, and none of those three papers addressed complexity.

The open case, in which evaluation is weak but terms are possibly open is strictly harder than the closed one, and close in spirit to the strong case, but easier to study—it is for instance the one studied by Grégoire and Leroy [20] when modeling (an old version of) the abstract machine of the kernel of Coq.

Open Call-by-Value. Open evaluation for CbV—shortened Open CbV—is particularly subtle because, as it is well-known, Plotkin's operational semantics is not adequate when dealing with open terms—see Accattoli and Guerrieri [8, 10]. Open CbV has been studied deeply by Accattoli and Guerrieri [8, 9, 10], Accattoli and Sacerdoti Coen [11], exploring different presentations, their rewriting, cost models, abstract machines, and denotational semantics. One of the motivations of this work is to add a new piece to the puzzle, by lifting the crumbling technique to the open case.

Our second contribution is to show that the crumbling technique smoothly scales up to Open CbV. We provide an abstract machine, the Open Crumble GLAM, and we show that it implements *the fireball calculus*—the simplest presentation of Open CbV—and that, as in the closed case, it only has a linear overhead. Two aspects of this study are worth pointing out. First, the technical development follows almost identically the one for the closed case, once the subtler invariants of the new machine have been found. Second, *the substitution of abstractions on demand*, a technical optimizations typical of open/strong cases (introduced in Accattoli and Dal Lago [7] and further studied in Accattoli and Guerrieri [9], Accattoli and Sacerdoti Coen [11]), becomes superfluous as it is subsumed by the crumbling transformation.

1.3 The Relationship with ANF

As long as one sticks to the untyped λ-calculus, crumbled forms coincide with ANF. The ANF, we said, is a variant of the CPS transformation. Roughly, the difference is that the ANF does not change the type, when terms are typed (here we work without types).

Kennedy [24] pointed out two problems with the ANF. One is the already discussed quadratic overhead, that does not affect our approach. The second one is the fact that the ANF does not smoothly scale up when the λ-calculus is extended to further constructs such as conditionals or pattern matching. Essentially, the ANF requires conditionals and pattern matching to be out of ES, that is, to never have an expression such as $s[x{\leftarrow}(\mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ u)]$. Unfortunately, these configurations can be created during evaluation. To preserve the ANF, one is led to add so-called *commuting conversions* such as:

$$s[x{\leftarrow}(\mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ u)] \to \mathsf{if}\ v\ \mathsf{then}\,(s[x{\leftarrow}t])\,\mathsf{else}\,(s[x{\leftarrow}u]) \qquad \text{(CC)}$$

Clearly, there is an efficiency issue: the commutation causes the duplication of the subterm s. A way out is to use a continuation-like technique, which makes Kennedy conclude that then there is no point in preferring ANF to CPS.

This is where our crumble representation departs from the ANF, as we do not require conditionals and pattern matching to be out of ES. Kennedy only studies the closed case. Our interest in open and strong evaluation is to explore the theory of implementation needed for proof assistants. In these settings, commutations of conditionals and pattern matching such as those hinted at by Kennedy are not valid: they are not validated by dependent type systems like those of Coq or Agda. For example, the CC rule above when the conditional is dependently typed breaks the property of *subject reduction*, as typed terms reduce to ill-typed terms. Consider the term:

$$(x+1)[x : (\mathsf{if}\ \mathit{true}\ \mathsf{then}\ \mathit{nat}\ \mathsf{else}\ \mathit{bool}){\leftarrow}\mathsf{if}\ \mathit{true}\ \mathsf{then}\ 0\ \mathsf{else}\ \mathit{false}] : \ \mathit{nat}$$

that has type *nat* because the type of x is convertible to *nat*. By applying rule CC, we obtain:

$$\mathsf{if}\ \mathit{true}\ \mathsf{then}\,((x+1)[x{\leftarrow}0])\ \mathsf{else}\,(\underline{(x+1)[x{\leftarrow}\mathit{false}]})$$

which is clearly ill-typed (in the underlined part).

The problem in the open case is actually more general, as not even the CPS would work: its properties do not scale up to open terms. In Section 9, indeed, we provide a counter-example to the simulation property in the open case.[2]

To sum up, neither commuting conversions nor the CPS transformation can be used in our framework. Therefore, we accept that conditionals and pattern matching may appear in ES (in contrast to Kennedy) and so depart from the ANF.

In the paper we treat the cases of the closed and open CbV calculi extended with conditionals. The essence of the study is the crumbling of β-reduction, not the conditionals. Conditionals are included only to stress the difference with respect to the ANF (pattern matching can be handled analogously), but they do not require a special treatment.

1.4 The Complexity of Abstract Machines

Asymptotic Bounds vs Benchmarking. The study of asymptotic bounds for abstract machines is meant to complement the use of benchmarking, by covering *all* possible cases, that certainly cannot be covered via benchmarking.

The relevance of such a study is evident when one considers open terms or strong evaluation. For strong evaluation, for instance, for more than 25 years in the literature there has been only Cregut's abstract machines [14], which on size exploding families of terms actually has exponential overhead (in the number of β-steps and the size of the initial term). A polynomial machine, developed via a careful asymptotic study, is in Accattoli [2]. Similarly, the abstract machine for open terms described in Grégoire and Leroy [20] suffers of exponential overhead on size exploding families (even if the authors then in practice implement a slightly different machine with polynomial overhead). The asymptotic study of this case is in Accattoli and Guerrieri [9], Accattoli and Sacerdoti Coen [11].

Abstract machines vs compilation. Abstract machines and compilation to machine language are two distinct techniques to execute a program. Compilation is typically more efficient, but it only handles the case where terms are closed and evaluation is weak, that is, the one of functional languages. Strong evaluation is sometimes

[2] Danvy and Filinski [17] claim that the CPS transformation scales up to open terms (their Theorem 2). However, as we discuss in Section 9, they consider only Plotkin's operational semantics, which is not adequate for open terms.

employed during compilation to optimize the compiled code, but typically only on linear redexes where size and time explosions are not an issue. Abstract machines are the only execution technique implemented in interactive theorem provers based on dependent types, that need strong evaluation.

Kennedy [24] argues that CPS-based translations are superior to ANF also because the CPS makes join points explicit as continuations, so that invocation of the continuation can be compiled efficiently using jumps. The argument is only valid for compilation and it does not affect abstract machines.

Garbage collection. We study *abstract* machines, which on purpose ignore many details of concrete implementations such as garbage collection, which is an orthogonal topic. In particular, garbage collection is always at most polynomial, if not linear, so its omission does not hide harmful blowups. As far as we know, no abstract machine implemented in interactive theorem provers performs garbage collection.

1.5 Related Work

Environments. In a recent work, Accattoli and Barras [5] compare various kinds of environments, namely, *global*, *local*, and *split*, from implementative and complexity points of view. The crumbling transformation can be studied with respect to every style of environment. Here we focus on crumbled *global* environments because they are simpler and because we also consider the open case, where all kinds of environment induce the same complexity.

Administrative Normal Forms. The literature on ANF is scarce. Beyond the already cited original papers, Danvy has also studied them and their relationship to CPS, but usually calling them *monadic normal forms* [15, 16, 23] because of their relationship with Moggi's monadic λ-calculus [29]. That terminology however sometimes describes a more liberal notion of terms, for instance in Kennedy [24], which is also another relevant piece in the literature on ANF.

All proofs and some supplementary material of our paper are in the Appendix of [6], the long version of this paper.

2 THE PIF CALCULUS

The grammars and the small-step operational semantics of the *Pif calculus* $\lambda^{\mathsf{if}}_{\mathsf{Plot}}$, that is, Plotkin's calculus λ_{Plot} [31] for Closed CbV evaluation extended with booleans and an if-then-else construct, plus error handling for clashing constructs, are in Fig. 1.

A term is either an application of two terms, an if-then-else, or a *value*, which is in turn either a variable, a (λ-)abstraction, true, false, or an error err. We distinguish values that are not variables, noted $v_{\neg x}$ and called *practical values*, following Accattoli and Sacerdoti Coen [12]. The *body* of an abstraction $\lambda x.t$ is t and the *bodies* of a conditional if t then u else s are its two branches u and s. Terms are always identified up to α-equivalence and the set of free variables of a term t is denoted by $\mathsf{fv}(t)$; t is *closed* if $\mathsf{fv}(t) = \emptyset$, *open* otherwise. We use $t\{x{\leftarrow}u\}$ for the term obtained by the capture-avoiding substitution of u for each free occurrence of x in t.

$$\begin{array}{rrcl}
\textsc{Terms} & t,u,s & ::= & v \mid tu \mid \mathsf{if}\ t\ \mathsf{then}\ u\ \mathsf{else}\ s \\
\textsc{Values} & v & ::= & x \mid v_{\neg x} \\
\textsc{Practical values} & v_{\neg x} & ::= & \lambda x.t \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{err} \\
\textsc{Right v-context} & R & ::= & \langle\cdot\rangle \mid tR \mid Rv \mid \mathsf{if}\ R\ \mathsf{then}\ u\ \mathsf{else}\ s
\end{array}$$

$$\textsc{Reduction Rules at Top Level}$$

$$\begin{array}{rcll}
(\lambda x.t)v & \mapsto_{\beta_v} & t\{x{\leftarrow}v\} & \\
\mathsf{if\ true\ then}\ t\ \mathsf{else}\ u & \mapsto_{\mathsf{ift}} & t & \\
\mathsf{if\ false\ then}\ t\ \mathsf{else}\ u & \mapsto_{\mathsf{iff}} & u & \\
\mathsf{if}\ t\ \mathsf{then}\ u\ \mathsf{else}\ s & \mapsto_{\mathsf{ife}} & \mathsf{err} & \mathsf{if}\ t = \lambda x.u\ \mathsf{or}\ t = \mathsf{err} \\
tu & \mapsto_{@\mathsf{e}} & \mathsf{err} & \mathsf{if}\ t \in \{\mathsf{true}, \mathsf{false}, \mathsf{err}\}
\end{array}$$

$$\textsc{Contextual closure}$$

$$\begin{array}{rcll}
R\langle t\rangle \to_a R\langle u\rangle & \mathsf{if} & t \mapsto_a u & \mathsf{for}\ a \in \{\beta_v, \mathsf{ift}, \mathsf{iff}, \mathsf{ife}, @\mathsf{e}\} \\
\to_{\mathsf{pif}} & := & \multicolumn{2}{l}{\to_{\beta_v} \cup \to_{\mathsf{ift}} \cup \to_{\mathsf{iff}} \cup \to_{\mathsf{ife}} \cup \to_{@\mathsf{e}}}
\end{array}$$

Figure 1: Pif calculus $\lambda^{\mathsf{if}}_{\mathsf{Plot}}$.

Contexts. In general, *contexts* are denoted by C and are terms with exactly one occurrence of a special constant $\langle\cdot\rangle$ called *the hole*, that is a placeholder for a removed subterm. In the paper we use various notions of contexts in different calculi—for $\lambda^{\mathsf{if}}_{\mathsf{Plot}}$ the relevant notion is *right (evaluation) v-context* R (see Fig. 1). The basic operation on (whatever notion of) contexts is the *plugging* $C\langle t\rangle$ of a term t for the hole $\langle\cdot\rangle$ in C: simply the hole is removed and replaced by t, possibly capturing variables.

Evaluation. According to the definition of right v-context, CbV evaluation $\to_{\mathsf{pif}}$ in $\lambda^{\mathsf{if}}_{\mathsf{Plot}}$ is *weak*, *i.e.* it does not reduce under λ-abstractions and in the branches of an if-then-else. CbV evaluation is defined for *any* (possibly open) term. But it is well-known that this operational semantics is *adequate* only for *closed* terms, as first noticed by Paolini and Ronchi Della Rocca [32], see also Accattoli and Guerrieri [8, 10], Guerrieri [21] and Guerrieri et al. [22]. When restricted to closed terms, $\lambda^{\mathsf{if}}_{\mathsf{Plot}}$ is called *Closed (Conditional) CbV*: in this setting, evaluation can fire a β-redex $(\lambda x.t)u$ only if the argument u is a closed value, *i.e.* a closed λ-abstraction, a boolean, or err; and in the production Rv for the definition of right v-contexts, v is always a closed value. Note that we work with right-to-left evaluation—this is forced by the production Rv in the definition of right evaluation v-contexts. In the closed case one could as well work with left-to-right evaluation, the choice is inessential.

The error constant err is generated during evaluation by the two cases of construct clashes: when the condition for an if-then-else is an abstraction and when a boolean is applied to a term. Both cases would be excluded by typing, but in our untyped setting they are possible, and handled via errors. Similarly, errors are also propagated when they appear as conditions for if-then-else and as left terms of an application. These cases are handled by rules $\to_{\mathsf{ife}}$ and $\to_{@\mathsf{e}}$. Note that errors do not propagate when they occur as arguments of applications: if the left sub-term of the application becomes an abstraction that erases the error then the error is *handled* and it is not observable.

A key property of Plotkin's Closed CbV is *harmony*: a closed term is β_v-normal if and only if it is a (closed) value *i.e.* a (closed) λ-abstraction. Therefore, every closed term either diverges or it evaluates to a (closed) λ-abstraction. Harmony extends to $\lambda^{\mathsf{if}}_{\mathsf{Plot}}$.

Proposition 2.1 (Pif harmony). *Let t be a closed term. t is $\to_{\mathsf{pif}}$-normal if and only if t is a value.*

3 CRUMBLED EVALUATION, INFORMALLY

Decomposing applications. The idea is to forbid the nesting of non-value constructs such as applications and if-then-else without losing expressive power. To ease the explanation, we focus on nested applications and forget about if-then-else—they do not pose any difficulty. Terms such as $(tu)s$ or $t(us)$ are then represented as $(\lambda x.(xs))(tu)$ and $(\lambda x.(tx))(us)$ where x is a fresh variable. It is usually preferred to use let expressions rather than introducing β-redexes, so that one would rather write let $x = tu$ in (xs) and let $x = us$ in (tx), or, with ES (aka environment entries),

$$(xs)[x{\leftarrow}tu] \quad \text{and} \quad (tx)[x{\leftarrow}us].$$

If the crumbling transformation $\underline{\cdot}$ is applied to the whole term—recursively on t, u and s in our examples—all applications have the form vv', *i.e.* they only involve values. If moreover CbV evaluation is adopted, then such a crumbled form is stable by evaluation (reduction steps are naturally defined so that a crumbled form reduces to a crumbled form), as variables can only be replaced by values.

Simulation and no evaluation contexts. Let us now have a look at a slightly bigger example and discuss the recursive part of the crumbling transformation. Let $I = \lambda x.x$ be the identity and consider the term $t \coloneqq ((\lambda y.yy)I)((II)I)$ whose right-to-left evaluation is

$$\begin{array}{llllll} t & \to_{\beta_v} & ((\lambda y.yy)I)(II) & \to_{\beta_v} & ((\lambda y.yy)I)I & \\ & \to_{\beta_v} & (II)I & \to_{\beta_v} & II & \to_{\beta_v} \quad I \end{array}$$

The crumbling transformation decomposes all applications, taking special care of grouping all the environment entries together, flattening them out (that is, avoiding having them nested one into the other), and reflecting the evaluation order in the arrangement of the environment. For instance, the crumbled representation $\underline{t}$ of the term t above is

$$\underline{t} = (wz)[w{\leftarrow}(\lambda y.yy)I][z{\leftarrow}xI][x{\leftarrow}II]$$

and evaluation takes always place at the end of the environment:

$$\begin{array}{lllllll} \underline{t} & \to_{\beta_v} & (wz)[w{\leftarrow}(\lambda y.yy)I][z{\leftarrow}xI][x{\leftarrow}I] & & & & \\ & \to_{[\,]} & (wz)[w{\leftarrow}(\lambda y.yy)I][z{\leftarrow}II] & & & & \\ & \to_{\beta_v} & (wz)[w{\leftarrow}(\lambda y.yy)I][z{\leftarrow}I] & & & & \\ & \to_{[\,]} & (wI)[w{\leftarrow}(\lambda y.yy)I] & \to_{\beta_v} & (wI)[w{\leftarrow}II] & & \\ & \to_{\beta_v} & (wI)[w{\leftarrow}I] & \to_{[\,]} & II & \to_{\beta_v} & I \end{array}$$

where the $\to_{\beta_v}$ steps correspond exactly to steps in the ordinary evaluation of t and $\to_{[\,]}$ steps simply eliminate the explicit substitution when its content is a value. Note how the transformation makes the redex always appear at the end of the environment, so that the need for searching for it—together with the notion of evaluation context—disappears.

Let us also introduce some terminology. Values and applications of values are *bites*. The transformation, called *crumbling translation*, turns a term into a pair, called *crumble*, of a bite and an environment.

Turning to micro-step evaluation. The previous example covers what happens when the crumbling transformation is paired with small-step evaluation. Abstract machines, however, employ a finer mechanism that we like to call *micro-step* evaluation, where the substitutions due to β-redexes are delayed and represented as new environment entries, and moreover substitution is decomposed as to act on one variable occurrence at a time. In particular, such a more parsimonious evaluation never removes environment entries because they might be useful later on—garbage collection is assumed to be an orthogonal and independent process. To give an idea of how micro steps work, let's focus on the evaluation of the subterm $(wz)[w{\leftarrow}(\lambda y.yy)I]$ of our example (because micro-step evaluations are long and tedious), that proceeds as follows:

$$\begin{array}{llll} (wz)[w{\leftarrow}(\lambda y.yy)I] & \to_{\beta_v} & (wz)[w{\leftarrow}yy][y{\leftarrow}I] & \to_{[\,]} \\ (wz)[w{\leftarrow}yI][y{\leftarrow}I] & \to_{[\,]} & (wz)[w{\leftarrow}II][y{\leftarrow}I] & \to_{\beta_v} \\ (wz)[w{\leftarrow}x][x{\leftarrow}I][y{\leftarrow}I] & \to_{[\,]} & (wz)[w{\leftarrow}I][x{\leftarrow}I][y{\leftarrow}I] & \to_{[\,]} \\ (Iz)[w{\leftarrow}I][x{\leftarrow}I][y{\leftarrow}I] & & & \end{array}$$

where $\to_{\beta_v}$ steps now introduce new environment entries. Now the redex is not always at the end of the environment, but it is always followed on the right by an environment whose entries are all abstractions, so that the search for the next redex becomes a straightforward visit from right to left of the environment—the evaluation context has been coded inside the sequential structure of the environment.

Abstraction bodies and the concatenation of environments. There is a last point to explain. We adopt weak evaluation—that only evaluates out of abstractions—but the crumbling transformation also transforms the bodies of abstractions and the branches of if-then-else into crumbles. Let us see another example. The crumbled representation of $u \coloneqq (\lambda x.((xx)(xx)))(II)$ then is

$$\underline{u} = ((\lambda x.((yz)[y{\leftarrow}xx][z{\leftarrow}xx]))w)[w{\leftarrow}II]$$

Micro-step evaluation goes as follows:

$$\begin{array}{lll} \underline{u} & \to_{\beta_v} & ((\lambda x.((yz)[y{\leftarrow}xx][z{\leftarrow}xx]))w)[w{\leftarrow}w'][w'{\leftarrow}I] \\ & \to_{[\,]} & ((\lambda x.((yz)[y{\leftarrow}xx][z{\leftarrow}xx]))w)[w{\leftarrow}I][w'{\leftarrow}I] \\ & \to_{[\,]} & ((\lambda x.((yz)[y{\leftarrow}xx][z{\leftarrow}xx]))I)[w{\leftarrow}I][w'{\leftarrow}I]. \end{array}$$

At this point, the reduction of the β-redex (involving λx) has to combine the crumble of the redex itself with the one of the body of the abstraction, by concatenating the environment of the former (here $[w{\leftarrow}I][w'{\leftarrow}I]$) at the end of the environment of the latter ($[y{\leftarrow}xx][z{\leftarrow}xx]$), interposing the entry created by the redex itself ($[x{\leftarrow}I]$), thus producing the new crumble:

$$(yz)[y{\leftarrow}xx][z{\leftarrow}xx][x{\leftarrow}I][w{\leftarrow}I][w'{\leftarrow}I].$$

The key conclusion is that evaluation needs to *concatenate* crumbled environments, which is an operation that ordinary abstract machines instead never perform.

Note that transforming abstraction bodies may produce nested ES, if the abstraction occurs in an ES. This is the only kind of nesting of ES that is allowed.

4 THE CRUMBLING TRANSFORMATION

In this section we formally define the language of crumbled forms and the crumbling transformation.

Crumbled forms. Terms are replaced by *crumbles*, which are formed by a bite and an environment, where in turn

- a *bite* is either a *crumbled value* (*i.e.* a variable, a boolean, an error, or an abstraction over a crumble), an application of crumbled values, or a if-then-else on a crumbled value whose alternatives are crumbles, and
- an *environment* is a finite sequence of explicit substitutions of bites for variables.

Formally, the definition is by mutual induction:

$$\begin{array}{rl}
\textsc{Bites} & b, b' ::= v \mid vw \mid \mathsf{if}\ v\ \mathsf{then}\ c\ \mathsf{else}\ d \\
\textsc{Crumbled values} & v, w ::= x \mid \lambda x.c \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{err} \\
\textsc{Environments} & e, e' ::= \epsilon \mid e[x{\leftarrow}b] \\
\textsc{Crumbles} & c, d ::= (b, e)
\end{array}$$

- ***Bodies*:** the bodies of abstractions and if-then-else are themselves crumbles—the forthcoming crumbling transformation is indeed *strong*, as it also transforms bodies.
- ***Crumbles are not closures*:** the definition of crumbles may remind one of *closures* in abstract machines with local environments, but the two concepts are different. The environment e of a crumble (b, e), indeed, does not in general bind all the free variables of the bite b.

We freely consider environments as lists extendable on both ends, and whose concatenation is obtained by simple juxtaposition. Given a crumble (b, e) and an environment e' the *appending* of e' to (b, e) is $(b, e)\ @\ e' := (b, ee')$.

Free variables, α-renaming, and all that. All syntactic expressions are *not* considered up to α-equivalence. Free variables are defined as expected for bites. For environments and crumbles they are defined as follows (via the auxiliary notion of *domain* of environments; this is because global environments are used here):

$$\begin{array}{rlrl}
\mathsf{dom}(e[x{\leftarrow}b]) & := \mathsf{dom}(e) \cup \{x\} & \mathsf{dom}(\epsilon) & := \emptyset \\
\mathsf{dom}((b, e)) & := \mathsf{dom}(e) & \mathsf{fv}(\epsilon) & := \emptyset \\
\mathsf{fv}(e[x{\leftarrow}b]) & := (\mathsf{fv}(e) \smallsetminus \{x\}) \cup \mathsf{fv}(b) & & \\
\mathsf{fv}((b, e)) & := (\mathsf{fv}(b) \smallsetminus \mathsf{dom}(e)) \cup \mathsf{fv}(e). & &
\end{array}$$

Let $e = [x_1{\leftarrow}b_1] \dots [x_k{\leftarrow}b_k]$ be an environment: we denote the lookup of x_i in e by $e(x_i) := b_i$. We say that a crumble c or an environment e are *well-named* if all the variables occurring on the lhs of ES outside abstractions in c or e are pairwise distinct.

The crumbling translation. A term is turned into a crumble via the following *crumbling translation* $\underline{\cdot}$, which uses an auxiliary translation $\overline{\cdot}$ from values into crumbled values.

$$\overline{x} := x \quad \overline{\lambda x.t} := \lambda x.\underline{t} \quad \overline{\mathsf{true}} := \mathsf{true} \quad \overline{\mathsf{false}} := \mathsf{false} \quad \overline{\mathsf{err}} := \mathsf{err}$$

$$\begin{array}{rlr}
\underline{v} & := (\overline{v}, \epsilon) & \underline{vv'} := (\overline{v}\overline{v'}, \epsilon) \\
\underline{tv} & := (x\overline{v}, [x{\leftarrow}b]e) & (*) \\
\underline{ut} & := \underline{ux}\ @\ ([x{\leftarrow}b]e) & (*) \\
\underline{\mathsf{if}\ v\ \mathsf{then}\ u\ \mathsf{else}\ s} & := (\mathsf{if}\ \overline{v}\ \mathsf{then}\ \underline{u}\ \mathsf{else}\ \underline{s}, \epsilon) & \\
\underline{\mathsf{if}\ t\ \mathsf{then}\ u\ \mathsf{else}\ s} & := (\mathsf{if}\ x\ \mathsf{then}\ \underline{u}\ \mathsf{else}\ \underline{s}, [x{\leftarrow}b]e) & (*)
\end{array}$$

$(*)$ if t is not a value and $\underline{t} = (b, e)$, and x is fresh.

According to the definition, if u and t are not values, $\underline{ut} = (yx, [y{\leftarrow}b']e'[x{\leftarrow}e])$ with $\underline{t} = (b, e)$, $\underline{u} = (b', e')$ and x, y fresh.

Example 4.1. Let $\delta := \lambda x.xx$ and $I := \lambda x.x$: thus, $\overline{I} = \lambda x.\underline{x} = \lambda x.(x, \epsilon)$ and $\overline{\delta} = \lambda x.\underline{xx} = \lambda x.(xx, \epsilon)$ (since $\underline{xx} = (xx, \epsilon)$) and $\underline{\delta\delta} = (\bar{\delta}\bar{\delta}, \epsilon)$. Therefore,

$$\begin{aligned}
\underline{\delta\delta I} &= (z\bar{I}, [z{\leftarrow}\bar{\delta}\bar{\delta}]) \\
&= (z\lambda x.(x, \epsilon), [z{\leftarrow}(\lambda x.(xx, \epsilon))\lambda x.(xx, \epsilon)])
\end{aligned}$$

$$\begin{aligned}
\underline{\delta\delta(xx)} &= (zw, [z{\leftarrow}\bar{\delta}\bar{\delta}][w{\leftarrow}xx]) \\
&= (zw, [z{\leftarrow}(\lambda x.(xx, \epsilon))\lambda x.(xx, \epsilon)][w{\leftarrow}xx]).
\end{aligned}$$

The crumbling translation $\underline{\cdot}$ is not surjective: the crumble $c := (xx, [x{\leftarrow}y])$ is such that $\underline{t} \neq c$ for any term t.

Read back. There is a left inverse for the crumbling translation, called *read-back* and defined by:

$$\begin{array}{rlrl}
x_{\downarrow} & := x & (\lambda x.c)_{\downarrow} & := \lambda x.c_{\downarrow} \\
\mathsf{true}_{\downarrow} & := \mathsf{true} & \mathsf{false}_{\downarrow} & := \mathsf{false} \\
\mathsf{err}_{\downarrow} & := \mathsf{err} & (vw)_{\downarrow} & := v_{\downarrow}w_{\downarrow} \\
(\mathsf{if}\ v\ \mathsf{then}\ c\ \mathsf{else}\ d)_{\downarrow} & := \mathsf{if}\ v_{\downarrow}\ \mathsf{then}\ c_{\downarrow}\ \mathsf{else}\ d_{\downarrow} & & \\
(b, e[x{\leftarrow}b'])_{\downarrow} & := (b, e)_{\downarrow}\{x{\leftarrow}b'_{\downarrow}\} & (b, \epsilon)_{\downarrow} & := b_{\downarrow}
\end{array}$$

Proposition 4.2 (Read-back and the crumbling translation). *For every term t and every value v, one has $\underline{t}_{\downarrow} = t$ and $\overline{v}_{\downarrow} = v$.*

Remark 4.1 (Crumbling translation, free variables).

(1) For any term t and any value v, one has $\mathsf{fv}(t) = \mathsf{fv}(\underline{t})$ and $\mathsf{fv}(v) = \mathsf{fv}(\overline{v})$; in particular, t is closed if and only if $\underline{t}$ is so.
(2) For any bite b and crumble c, $\mathsf{fv}(b_{\downarrow}) = \mathsf{fv}(b)$ and $\mathsf{fv}(c_{\downarrow}) = \mathsf{fv}(c)$.
(3) The crumbling translation commutes with the renaming of free variables.
(4) The crumbling translation and the read-back map values to values.

Crumbled contexts. For crumbled forms, we need *contexts* both for environments and crumbles:

$$\begin{array}{rl}
\textsc{Environment contexts} & E := e[x{\leftarrow}\langle\cdot\rangle] \\
\textsc{Crumble contexts} & C := \langle\cdot\rangle \mid (b, E)\,.
\end{array}$$

Crumbles can be plugged into both notions of contexts. Let us point out that the following definition of plugging is slightly unusual as it does a little bit more than just replacing the hole, because simply replacing would not provide a well-formed syntactic object: plugging indeed extracts the environment from the plugged crumble and concatenates it with the environment of the context. Such an unusual operation—that may seem *ad-hoc*—is actually one of the key technical points in order to obtain a clean proof of the implementation theorem (see Section 5.2).

Definition 4.3 (Plugging in crumbled contexts). Let $E = e[x{\leftarrow}\langle\cdot\rangle]$ be an environment context, C be a crumble context, and $c = (b', e')$ be a crumble. The *plugging* $E\langle c\rangle$ of c in E and the *plugging* $C\langle c\rangle$ of c in C are defined by

$$(e[x{\leftarrow}\langle\cdot\rangle])\langle(b', e')\rangle := e[x{\leftarrow}b']e' \quad \langle\cdot\rangle\langle c\rangle := c \quad (b, E)\langle c\rangle := (b, E\langle c\rangle)$$

Example 4.4. In Example 4.1 we have seen that $\underline{\delta\delta I} = (z\lambda x.x_{\epsilon}, [z{\leftarrow}(\lambda x.(xx)_{\epsilon})\lambda x.(xx)_{\epsilon}])$, where we set $b_{\epsilon} := (b, \epsilon)$ for any bite b. We have that $\underline{\delta\delta I} = C\langle c\rangle$ with $C := (z\lambda x.x_{\epsilon}, [z{\leftarrow}\langle\cdot\rangle])$ and $c := ((\lambda x.(xx)_{\epsilon})\lambda x.(xx)_{\epsilon}, \epsilon)$.

The notions of well-named, $\mathsf{fv}(\cdot)$, and $\mathsf{dom}(\cdot)$ can be naturally extended to crumble contexts. The definition of read back is extended to crumble contexts by setting $\langle\cdot\rangle_{\downarrow} := \langle\cdot\rangle$ and $(b, e[x{\leftarrow}\langle\cdot\rangle])_{\downarrow} := (b, e)_{\downarrow}\{x{\leftarrow}\langle\cdot\rangle\}$. Note however that the unfolding of a crumble context is not necessarily a context, because the hole can be duplicated or erased by the unfolding. For instance, let $C := (x\,x, [x{\leftarrow}\langle\cdot\rangle])$. Then $C_{\downarrow} = \langle\cdot\rangle\langle\cdot\rangle$ is not a context.

Lemma 4.5 provides the properties of the translation needed to prove the invariants of machines in the next sections.

Lemma 4.5 (Properties of crumbling). *For every term t:*

(1) Freshness: *$\underline{t}$ is well-named.*
(2) Closure: *if t is closed, then* $\mathsf{fv}(\underline{t}) = \emptyset$.
(3) Disjointedness*:* $\mathsf{dom}(C) \cap \mathsf{fv}(b) = \emptyset$ *if $\underline{t} = C\langle(b, e)\rangle$.*
(4) Bodies: *every body in $\underline{t}$ is the translation of a term.*
(5) Contextual decoding: *if $\underline{t} = C\langle c\rangle$, then $C_{\downarrow}$ is a right v-context.*

5 THE CLOSED CASE

Here we show how to evaluate crumbled forms with a micro-step operational semantics. We builds over the work of Accattoli and co-authors, who employ the following terminology:

- *Calculus*: for a small-step semantics where both substitution and search for the redex are meta-level operations;
- *Linear calculus*: for a micro-step semantics where substitution is decomposed—the calculus has ES and possibly a notion of environment if the ES are grouped together—but the search for the redex is still meta-level and expressed via evaluation contexts;
- *Abstract machine*: for a micro-step semantics where both substitution and search for the redex are decomposed. The search for redexes is handled via one or more stacks called *applicative stack*, *dump*, *frame*, and so on; the management of names is also explicit, *i.e. not* up-to α-equivalence.

The crumbling transformation blurs the distinction between a linear calculus and an abstract machine because it allows using the sequential structure of the environment as the only stack needed to search for redexes.

The operational semantics for crumbled forms we present next is in the style of a linear calculus, because spelling out the straightforward search for redexes is not really informative. Nonetheless, we do call it an *abstract machine*, because of the blurred distinction in the crumble case and because we manage names explicitly. In Section 7 we sketch the actual abstract machine (details are in [6]).

5.1 The Crumble GLAM

Transitions. To introduce the Crumble GLAM (GLAM stands for Global Leroy Abstract Machine) we need some definitions. First, environments and crumbles made out of practical values only are defined and noted as follows:

$$\begin{array}{rl} v\text{-ENVIRONMENTS} & e_v ::= \epsilon \mid e_v[x{\leftarrow}v_{\neg x}] \\ v\text{-CRUMBLES} & c_v ::= (v_{\neg x}, e_v) \end{array}$$

Essentially, a v-environment stands for the already evaluated coda of the environment described in the paragraph about micro-steps in Sect. 3, while v-crumbles are fully evaluated crumbles (*i.e.* final states of the machine), as we show below.

Second, given a crumble c we use c^{α} for a crumble obtained by α-renaming the names in the domain of c with fresh ones so that c^{α} is well-named.

The transitions act on crumbles whose environments are v-environments. The top level transitions are:

$$((\lambda x.c)\, v, e_v) \mapsto_{\beta_v} (c\ @\ [x{\leftarrow}v])^{\alpha}\ @\ e_v$$

$$\begin{array}{rll} (\mathsf{if\,true\,then}\,c\,\mathsf{else}\,d, e_v) & \mapsto_{\mathsf{ift}} c\ @\ e_v & \\ (\mathsf{if\,false\,then}\,c\,\mathsf{else}\,d, e_v) & \mapsto_{\mathsf{iff}} d\ @\ e_v & \\ (\mathsf{if}\,v\,\mathsf{then}\,c\,\mathsf{else}\,d, e_v) & \mapsto_{\mathsf{ife}} (\mathsf{err}, e_v) & (1) \\ (vw, e_v) & \mapsto_{@\mathsf{e}} (\mathsf{err}, e_v) & (2) \\ (x, e_v) & \mapsto_{\mathsf{sub}_{var}} (e_v(x), e_v) & (3) \\ (xv, e_v) & \mapsto_{\mathsf{sub}_l} (e_v(x)\, v, e_v) & (3) \\ (\mathsf{if}\,x\,\mathsf{then}\,c\,\mathsf{else}\,d, e_v) & \mapsto_{\mathsf{sub_{if}}} (\mathsf{if}\,e_v(x)\,\mathsf{then}\,c\,\mathsf{else}\,d, e_v) & (3) \end{array}$$

(1) if $v = \lambda x.e$ or $v = \mathsf{err}$
(2) if $v \in \{\mathsf{true}, \mathsf{false}, \mathsf{err}\}$
(3) if $x \in \mathsf{dom}(e_v)$

Transitions are then closed by crumble contexts: for every $a \in \{\beta_v, \mathsf{ift}, \mathsf{iff}, \mathsf{ife}, @\mathsf{e}, \mathsf{sub}_{var}, \mathsf{sub}_l, \mathsf{sub_{if}}\}$ define $C\langle c\rangle \to_a C\langle d\rangle$ if $c \mapsto_a d$. The transition relation $\to_{\mathsf{Cr}}$ of the Crumble GLAM is defined as the union of all these rules. Let us explain each transition:

- $\to_{\beta_v}$: (forget about the α-renaming for the moment—see the next paragraph) the rule removes a β-redex and introduces an ES $[x{\leftarrow}v]$ instead of performing the meta-level substitution. Moreover, the environment of the body c of the abstraction and the external environment e_v are concatenated (via the appending operation @) interposing $[x{\leftarrow}v]$.
- *Conditional and error transitions* $\to_{ift}, \to_{iff}, \to_{ife}, \to_{@e}$: these transitions simply mimics the analogous rules on the Pif calculus, with no surprises.
- *Substitution transitions* $\to_{\mathsf{sub}_l}, \to_{\mathsf{sub}_y}, \to_{\mathsf{sub_{if}}}$: the variable x is substituted by the corresponding crumbled value in the environment e_v, if any. In the closed case, a forthcoming invariant guarantees that $e_v(x)$ is always defined so that side-condition (3) is actually always satisfied. There are no rules to substitute on the right of an application (see below).

According to the definitions of plugging and top level transitions, the transition relation follows *right-to-left* evaluation, as the environment on the right of a redex is a v-environment (made of practical values only), which means that it has already been evaluated (see the harmony property for Crumble GLAM in Prop. 5.3 below). Adopting right-to-left evaluation implies that the Crumble GLAM does not need a rule $\to_{\mathsf{sub}_r}$ symmetrical to $\to_{\mathsf{sub}_l}$, whose top level shape would be $(vx, e_v) \mapsto_{\mathsf{e}_r} (v\, e_v(x), e_v)$ with $x \in \mathsf{dom}(e_v)$: indeed, if v is a variable then $\to_{\mathsf{sub}_l}$ applies to the same redex (vx, e_v), otherwise v is an abstraction and $\to_{\beta_v}$ applies to (vx, e_v).

The cost and the place of α-renaming. Abstract machines with global environments have to α-rename at some point, this is standard[3]. In our implementation, renaming is implemented as a copy function. And the cost of renaming is under control because of forthcoming invariants of the machine. This is all standard [5].

[3] Local environments do allow to avoid renamings, but the simplification is an illusion, as the price is payed elsewhere—see Accattoli and Barras [5]—there is no real way out.

Often the burden of renaming/copying is put on the substitution rules. It is less standard to put it on the β_v-transition, as we do here, but nothing changes. Last, a technical remark: in rule $\to_{\beta_v}$ the α-renaming at top level has to pick names that are fresh also with respect to the crumble context enclosing it. This point may seem odd but it is necessary to avoid name clashes, and it is trivially obtained in our concrete implementation, where variable names are memory locations and picking a fresh name amounts to allocating a new location, that is of course new globally.

Definition 5.1 (Reachable crumble). A crumble is *reachable* (by the Crumble GLAM) if it is obtained by a sequence of transitions starting from the translation $\underline{t}$ of a closed term t.

Unchaining abstractions. The substitution performed by the rule $\to_{\mathsf{sub}_{var}}$ may seem an unneeded optimization; quite the opposite, it fixes an issue causing quadratic overhead in the machine. The culprits are malicious *chains of renamings*, *i.e.* environments of the form $[x_1{\leftarrow}x_2][x_2{\leftarrow}x_3]\cdots[x_n{\leftarrow}\lambda y.c]$ substituting variables for variables and finally leading to an abstraction. Accattoli and Sacerdoti Coen [11] showed that the key to linear overhead is to perform substitution steps while going through the chain from right to left.

Example 5.2. Consider the crumble $\underline{\delta\delta} = (\overline{\delta}\,\overline{\delta}, \epsilon)$, where $\overline{\delta} = \lambda x.(xx, \epsilon)$; then:

$$\begin{aligned}\underline{\delta\delta} &\to_{\beta_v} (xx, [x{\leftarrow}\overline{\delta}]) \qquad \to_{\mathsf{sub}_l} (\overline{\delta}\,x, [x{\leftarrow}\overline{\delta}])\\ &\to_{\beta_v} (yy, [y{\leftarrow}x][x{\leftarrow}\overline{\delta}]) \to_{\mathsf{sub}_{var}} (yy, [y{\leftarrow}\overline{\delta}][x{\leftarrow}\overline{\delta}]) \to_{\mathsf{sub}_l} \cdots\end{aligned}$$

In Ex. 4.1 we introduced the crumble $\underline{\delta\delta I} = (z\,\overline{I}, [z{\leftarrow}\overline{\delta}\,\overline{\delta}])$ where $\overline{I} = (\lambda x.(x, \epsilon))$; in accordance with the crumble decomposition shown in Ex. 4.4, we have:

$$\begin{aligned}\underline{\delta\delta I} \quad &\to_{\beta_v} (z\,\overline{I}, [z{\leftarrow}xx][x{\leftarrow}\overline{\delta}]) \to_{\mathsf{sub}_l} (z\,\overline{I}, [z{\leftarrow}\overline{\delta}\,x][x{\leftarrow}\overline{\delta}])\\ &\to_{\beta_v} (z\,\overline{I}, [z{\leftarrow}yy][y{\leftarrow}x][x{\leftarrow}\overline{\delta}])\\ &\to_{\mathsf{sub}_l} (z\,\overline{I}, [z{\leftarrow}yy][y{\leftarrow}\overline{\delta}][x{\leftarrow}\overline{\delta}]) \to_{\mathsf{sub}_l} \cdots\end{aligned}$$

Consider now the *open* crumble

$$c \coloneqq \underline{\delta\delta(xx)} = (zw, [z{\leftarrow}\overline{\delta}\,\overline{\delta}][w{\leftarrow}xx]).$$

The crumble c is normal because its only possible decomposition of the form $C\langle(b, e_v)\rangle$ is for $e_v = \epsilon$ (as xx is not a practical value), and no transitions apply to the rightmost entry $[w{\leftarrow}xx]$ since x is free.

The Crumble GLAM satisfies a harmony property.

Proposition 5.3 (Harmony for the Crumble GLAM). *A closed crumble c is normal if and only if it is a v-crumble.*

5.2 The Implementation Theorem

To show that the Crumble GLAM correctly implements the Pif calculus, we apply an abstract approach introduced by Accattoli and Guerrieri [9], which we reuse as well in the following sections for other crumble abstract machines and other evaluation strategies of the λ-calculus.

The implementation theorem, abstractly. In Accattoli and Guerrieri [9] it is proven that, given

- a generic abstract machine M, which is a *transitions relation* $\rightsquigarrow_{\mathsf{M}}$ over a set of *states* that splits into
  - *principal transitions* $\rightsquigarrow_{\mathrm{p}}$, that corresponds to the evaluation steps on the calculus, and
  - *overhead transitions* $\rightsquigarrow_{\mathrm{o}}$, that are specific of the machine,
- an evaluation strategy $\to$ in the λ-calculus, and
- a decoding $(\cdot)_{\downarrow}$ of states of M into terms,

M correctly implements $\to$ via $(\cdot)_{\downarrow}$ whenever $(\mathsf{M}, \to, (\cdot)_{\downarrow})$ forms an *implementation system*, *i.e.* whenever the following conditions are fulfilled (where s and s' stand for generic states of M):

(1) *Initialization:* there is an encoding $\underline{\cdot}$ of terms such that $\underline{t}_{\downarrow} = t$;
(2) *Principal projection*: $s \rightsquigarrow_{\mathrm{p}} s'$ implies $s_{\downarrow} \to s'_{\downarrow}$;
(3) *Overhead transparency*: $s \rightsquigarrow_{\mathrm{o}} s'$ implies $s_{\downarrow} = s'_{\downarrow}$;
(4) *Determinism*: $\rightsquigarrow_{\mathsf{M}}$ is deterministic;
(5) *Halt*: M final states (to which no transition applies) decode to $\to$-normal terms;
(6) *Overhead Termination*: $\rightsquigarrow_{\mathrm{o}}$ terminates.

Our notion of implementation, tuned towards complexity analyses, requires a perfect match between the number of steps of the strategy and the number of principal transitions of the execution.

Theorem 5.4 (Machine implementation, [9]). *If a machine M, a strategy $\to$ on λ-terms and a decoding $\cdot_{\downarrow}$ form an implementation system then:*

(1) Executions to derivations*: for any M-execution $\rho\colon \underline{t} \rightsquigarrow_{\mathsf{M}}^{*} s$ there is a $\to$-derivation $d\colon t \to^{*} s_{\downarrow}$.*
(2) Derivations to executions*: for every $\to$-derivation $d\colon t \to^{*} u$ there is an M-execution $\rho\colon \underline{t} \rightsquigarrow_{\mathsf{M}}^{*} s$ such that $s_{\downarrow} = u$.*
(3) Principal matching*: in both previous points the number $|\rho|_{\mathrm{p}}$ of principal transitions in ρ is exactly the length $|d|$ of the derivation d, i.e. $|d| = |\rho|_{\mathrm{p}}$.*

The crumbling implementation system. The states of the Crumble GLAM are crumbles. Its principal transitions are those labeled with $\{\beta_v, \mathsf{ift}, \mathsf{iff}, \mathsf{ife}, @\mathsf{e}\}$, while the overhead transitions are those labeled with $\{\mathsf{sub}_{var}, \mathsf{sub}_l, \mathsf{sub}_{\mathsf{if}}\}$. We can now show that the Crumble GLAM, Pif evaluation $\to_{\mathsf{pif}}$ and the read-back $(\cdot)_{\downarrow}$ form an implementation system, that is, that the Crumble GLAM implements the Pif calculus.

We are going to provide five of the six sufficient conditions required by the implementation theorem (Thm. 5.4); the sixth one, the termination of overhead transitions, is subsumed by the finer complexity analysis in Subsect. 5.3.

The sufficient conditions, as usual, are proved by means of a few invariants of the machine, given by Lemma 5.5 below. These invariants are essentially the properties of the translation in Lemma 4.5 extended to all reachable crumbles. One of them—namely *contextual decoding*—however, is weaker because reachable crumbles do not necessarily have the same nice structure as the initial crumbles obtained by translation of a term, as the next remark explains.

Remark 5.1. Even though not all crumble contexts unfold to contexts, crumble contexts obtained by decomposing crumbles given by the translation of terms do (Lemma 4.5.5)—this is the contextual decoding property. Unfortunately, it is not preserved by evaluation. Consider the crumble $c \coloneqq \underline{(\lambda x.x(xx))\,I} = ((\lambda x.(xy, [y{\leftarrow}xx]))\overline{I}, \epsilon)$ with $\overline{I} = \lambda z.(z, \epsilon)$. Clearly, $c = \langle\underline{(\lambda x.x(xx))\,I}\rangle$ where $\langle\cdot\rangle_{\downarrow} = \langle\cdot\rangle$ is a context. After one β_v step, the crumble c reaches

$(xy, [y{\leftarrow}xx][x{\leftarrow}\bar{I}]) = C\langle(\bar{I}, \epsilon)\rangle$ for $C := (xy, [y{\leftarrow}xx][x{\leftarrow}\langle\cdot\rangle])$. But C unfolds to $C_{\downarrow} = \langle\cdot\rangle(\langle\cdot\rangle\langle\cdot\rangle)$, which is not a λ-context.

Lemma 5.5 (Invariants for the Crumble GLAM). *For every reachable crumble c in the Crumble GLAM:*

(1) Freshness: *c is well-named.*
(2) Closure: $\mathsf{fv}(c) = \emptyset$.
(3) Bodies: *every body occurring in c is a subterm (up to renaming) of the initial crumble.*
(4) Weak contextual decoding: *for every decomposition $C\langle(b, e_v)\rangle$ where b is not a crumbled value, if C'' is a prefix of C then $C''_{\downarrow}$ is a right v-context.*

Freshness and closure are invariants needed to ensure the basic functioning of the machine. The bodies invariant corresponds to what is often called *subterm invariant*: it is the key invariant for complexity analyses, as it allows to bound the size of duplicated subterms (that are always abstractions) using the size of the initial term. Usually, it is only needed for complexity analyses, while here it is needed for the implementation theorem too (namely, only for the proof of the weak contextual decoding invariant). The weak contextual decoding invariant is crucial to show that principal transitions of the Crumble GLAM project on evaluation steps in $\lambda^{\mathsf{if}}_{\mathsf{Plot}}$.

Theorem 5.6 (Implementation). *Let c be a crumble reachable by the Crumble GLAM.*

(1) Initialization: *$\underline{t}_{\downarrow} = t$ for every term t.*
(2) Principal projection: *if $c \to_a d$ then $c_{\downarrow} \to_a d_{\downarrow}$, for any rule $a \in \{\beta_v, ift, iff, ife, @e\}$.*
(3) Overhead transparency: *if $c \to_a d$ then $c_{\downarrow} = d_{\downarrow}$ for any rule $a \in \{\mathsf{sub}_{var}, \mathsf{sub}_l, \mathsf{sub}_{\mathsf{if}}\}$.*
(4) Determinism: *the transition $\to_{\mathsf{Cr}}$ is deterministic.*
(5) Halt: *if c is $\to_{\mathsf{Cr}}$-normal then $c_{\downarrow}$ is $\to_{\mathsf{pif}}$-normal.*
(6) Overhead termination: *$\to_a$ terminates, for any rule $a \in \{\mathsf{sub}_{var}, \mathsf{sub}_l, \mathsf{sub}_{\mathsf{if}}\}$.*

Therefore, the Crumble GLAM, Pif evaluation $\to_{\mathsf{pif}}$, and the read-back $(\cdot)_{\downarrow}$ form an implementation system.

5.3 Complexity for the closed case

To estimate the cost of the Crumble GLAM, we provide first an upper bound on the number of overhead transitions—namely the substitution ones sub_{var}, sub_l, and $\mathsf{sub}_{\mathsf{if}}$—in an execution ρ as a function of the number $|\rho|_{\mathsf{p}}$ of principal transitions. Then we discuss the cost of implementing single transitions. Last, by composing the two analyses we obtain the total cost, that is linear in the number of principal transitions and in the size of the initial term/crumble, that is, the machine is bilinear.

Number of transitions: non-renaming substitutions. Let $\rho : c_0 \to^*_{\mathsf{Cr}} c$ be an execution (*i.e.* a sequence of transitions) in the Crumble GLAM and let $|\rho|_{\mathsf{p}}, |\rho|_{\mathsf{sub}_{var}}, |\rho|_{\mathsf{sub}_l}, |\rho|_{\mathsf{sub}_{\mathsf{if}}}$ be the number of principal, sub_{var}, sub_l, and $\mathsf{sub}_{\mathsf{if}}$ transitions in ρ, respectively. Clearly, a sub_l transition can only be immediately followed by a β_v or a @e transition (since $\to_{\mathsf{Cr}}$ is deterministic), and so $|\rho|_{\mathsf{sub}_l} \leq |\rho|_{\beta_v} + |\rho|_{@\mathsf{e}} + 1$. Similarly, a $\mathsf{sub}_{\mathsf{if}}$ transition is immediately followed by a ift, a iff or a ife transition. Therefore, $|\rho|_{\mathsf{sub}_l} + |\rho|_{\mathsf{sub}_{\mathsf{if}}} \leq |\rho|_{\mathsf{p}} + 1$.

Number of transitions: renaming steps. The analysis for $|\rho|_{\mathsf{sub}_{var}}$ is subtler. A *variable crumble* is a crumble of the form (x, e). The number of sub_{var} transitions is bounded by the number of variable crumbles out of bodies appearing in evaluation position along an execution $\rho\colon c_0 \to^*_{\mathsf{Cr}} c$. These can be due to the following reasons:

(1) *Static*: variable crumbles out of bodies in the initial state c_0;
(2) *Dynamic*: variable crumbles obtained dynamically. In turn, these are divided into (see also the discussion after Prop. 5.7):
   (a) *Copy*: variable crumbles occurring in the bodies of abstractions and if-then-else (and thus frozen) that become active because the construct is evaluated and the body exposed;
   (b) *Creation*: variable crumbles that cannot be traced back to variable crumbles appearing in prefixes of the execution.

We now show that the crumbling translation does not produce any variable crumbles out of bodies, but one, if the original term is itself a variable. Therefore, the contribution of point 1 is at most 1. We need a measure, counting variable crumbles out of bodies. Note that a variable crumble (x, e) appearing in a crumble context C rather takes the form $[y{\leftarrow}x]e$, which is why the following measure counts the substitutions containing only a variable.

$$\begin{array}{lll} |b|_{\mathrm{var}} := 0 & \text{if } b \text{ is not a variable} & \\ |x|_{\mathrm{var}} := 1 & & |(b, e)|_{\mathrm{var}} := |b|_{\mathrm{var}} + |e|_{\mathrm{var}} \\ |\epsilon|_{\mathrm{var}} := 0 & & |e[x{\leftarrow}b]|_{\mathrm{var}} := |e|_{\mathrm{var}} + |b|_{\mathrm{var}}\,. \end{array}$$

Proposition 5.7. *Let t be a term and v a value. Then:*

(1) $|\underline{t}|_{\mathrm{var}} \leq 1$*; and* $|\underline{t}|_{\mathrm{var}} = 1$ *if and only if t is a variable;*
(2) $|\overline{v}|_{\mathrm{var}} \leq 1$*; and* $|\overline{v}|_{\mathrm{var}} = 1$ *if and only if v is a variable.*

Let us now discuss the variable crumbles of point 2.a (dynamic copy). By the bodies invariant (Lemma 5.5.3), these pairs appear in a body of the initial crumble. By the bodies property of the crumbling translation (Lemma 4.5.4), all these bodies are the translation of a term, and—by using Prop. 5.7 again—we obtain that each such body contributes at most with one variable crumble. Since each body is exposed by one $\to_{\beta_v}$ or $\to_{\mathsf{ift}}$ or $\to_{\mathsf{iff}}$ transition, we have that the variable crumbles of point 2.a are bounded by $|\rho|_{\mathsf{p}}$.

Last, we bound the number of variable crumbles at point 2.b (dynamic creation). There is only one rule that can create a new variable crumble (and exactly one), namely $\to_{\beta_v}$ when the argument of the β-redex is a variable. For instance,

$$((\lambda x.(xx, \epsilon))y, [y{\leftarrow}\lambda z.z]) \to_{\beta_v} (xx, [x{\leftarrow}y][y{\leftarrow}\lambda z.z])$$

where the created variable crumble is $(y, [y{\leftarrow}\lambda z.z])$. Then the number of variable crumbles at point 2.b is bounded by the number of $\to_{\beta_v}$ transitions, itself bounded by $|\rho|_{\mathsf{p}}$.

The following lemma sums up the previous discussions

Lemma 5.8. *Let $\rho : c_0 \to^*_{\mathsf{Cr}} c$ be a Crumble GLAM execution.*

(1) Linear number of non-renamings substitutions*:* $|\rho|_{\mathsf{sub}_l} + |\rho|_{\mathsf{sub}_{\mathsf{if}}} \leq |\rho|_{\mathsf{p}} + 1$.
(2) Linear number of renamings*:* $|\rho|_{\mathsf{sub}_{var}} \leq 2|\rho|_{\mathsf{p}} + 1$.
(3) Linear number of substitutions*:* $|\rho|_{\mathsf{sub}_l} + |\rho|_{\mathsf{sub}_{var}} + |\rho|_{\mathsf{sub}_{\mathsf{if}}} \leq 3|\rho|_{\mathsf{p}} + 2$.

Cost of single transitions. Performing a single transition $\to$ in the Crumble GLAM consists of four operations:

(1) *Search*: locating the next redex;

(2) *Unplugging*: splitting the crumble to be reduced into a crumble context C and the crumble c that is the redex at top level;
(3) *Rewriting*: applying a rewriting rule to the crumble c, obtaining a new crumble d;
(4) *Plugging*: putting the new crumble back into the crumble context obtaining $C\langle d\rangle$.

The search for redexes is embedded into the definition of the rules, via the contextual closure. The technical definition of plugging and unplugging of crumbles into a crumble context is quite involved and, if implemented literally, is not constant time.

To ease the reasoning, in this section we assume that search and (un)plugging have negligible cost and show that the total cost of rewriting is bilinear. In Section 7 we sketch a slight variant of the Crumble GLAM, the Pointed Crumble GLAM, that adds a transition for searching redexes and removes the need for plugging and unplugging (details are in [6]). A further analysis of the Pointed Crumble GLAM shows that the total cost of search and (un)plugging is bilinear and thus negligible, justifying the results of this section.

Cost of single transitions: β_v transitions. We denote by $|t|$, $|c|$, $|e|$ and $|b|$ the size of terms, crumbles, environments and bites, respectively, defined as follows:

$$\begin{aligned} |x| &:= 1 & |tu| &:= |t| + |u| + 1 \\ |\mathsf{true}| = |\mathsf{false}| &:= 1 & |\lambda x.t| &:= |t| + 1 \\ |\mathsf{if}\ t\ \mathsf{then}\ u\ \mathsf{else}\ s| &:= |t| + |u| + |s| + 1 & |\mathsf{err}| &:= 1 \\ |\epsilon| &:= 0 & |e[x{\leftarrow}b]| &:= |e| + |b| \\ |(b, e)| &:= |b| + |e|. \end{aligned}$$

The cost of each β_v transition (that needs to perform a copy of the crumble in the abstraction in order α-rename it) is bound by the size of the copied crumble. By the bodies invariant (Lemma 5.5.3) the abstraction is the α-renaming of one the abstractions already present in the initial crumble. Therefore the cost of a β_v transition is bound by the size of the initial crumble. The next lemma shows that the size of the initial crumble is linear in the size of the initial term translating to the crumble. Therefore, the cost of a β_v transition is linear by the size of the initial term.

Lemma 5.9 (Size of translated terms). *Let t be a term and v a value. Then $|\underline{t}| \leq 5|t|$ and $|\overline{v}| \leq 5|v|$.*

Cost of single transitions: substitutions. The cost of sub_l, sub_{var}, and $\mathsf{sub}_{\mathsf{if}}$ transitions depends on the choice of data structures for implementing the machine. Following the literature on global environment machines [5], we assume the global environment to be implemented as a store and variable occurrences to be implemented as pointers into the store, so that lookup in the environment can be performed in constant time on a Random Access Machine (RAM). As for the cost of actually performing the replacement of x with $e_v(x)$ in the sub_{var}, sub_l and $\mathsf{sub}_{\mathsf{if}}$ rules, it can be done in constant time by copying the pointer to $e_v(x)$. This is possible because the actual copy, corresponding to α-renaming, is done in the β_v step. Thus, single substitution transitions have constant cost.

Cost of single transitions: conditionals and errors. It is immediate that—if one excludes plugging and unplugging—these transitions have constant cost.

$$\begin{array}{rrcl} \textsc{Terms} & t, u & ::= & \ldots\ \ (\text{as in } \lambda^{\mathsf{if}}_{\mathsf{Plot}}, \text{ see Figure 1}) \\ \textsc{Values} & v & ::= & \ldots\ \ (\text{as in } \lambda^{\mathsf{if}}_{\mathsf{Plot}}, \text{ see Figure 1}) \\ \textsc{Fireballs} & f & ::= & v \mid i \\ \textsc{Inert terms} & i & ::= & x\,f \mid i\,f \mid \mathsf{if}\ x\ \mathsf{then}\ t\ \mathsf{else}\ u \\ & & & \mid \mathsf{if}\ i\ \mathsf{then}\ t\ \mathsf{else}\ u \\ \textsc{Right f-context} & R & ::= & \langle\cdot\rangle \mid tR \mid Rf \mid \mathsf{if}\ R\ \mathsf{then}\ u\ \mathsf{else}\ s \end{array}$$

$$\textsc{Reduction Rules at Top Level}$$

$$(\lambda x.t)i \mapsto_{\beta_i} t\{x{\leftarrow}i\} \qquad \mapsto_{\beta_v}, \mapsto_{\mathsf{ift}}, \mapsto_{\mathsf{iff}}, \mapsto_{\mathsf{ife}}, \mapsto_{@\mathsf{e}} \ \text{ as in } \lambda^{\mathsf{if}}_{\mathsf{Plot}}$$

$$\textsc{Contextual closure}$$

$$R\langle t\rangle \to_a R\langle u\rangle \quad \text{if} \quad t \mapsto_a u \quad \text{for } a \in \{\beta_v, \beta_i, \mathsf{ift}, \mathsf{iff}, \mathsf{ife}, @\mathsf{e}\}$$

$$\to_{\beta_f} := \to_{\beta_v} \cup \to_{\beta_i} \qquad \to_{\mathsf{c}\beta_f} := \bigcup_{a \in \{\beta_f, \mathsf{ift}, \mathsf{iff}, \mathsf{ife}, @\mathsf{e}\}} \to_a$$

Figure 2: The conditional fireball calculus $\lambda^{\mathsf{if}}_{\mathsf{fire}}$.

Cost of executions. Summing up all the analyses in this section we obtain the following theorem.

Theorem 5.10 (The Crumble GLAM is bilinear up to search and (un)plugging). *For any closed term t and any Crumble GLAM execution $\rho\colon \underline{t} \to^*_{\mathsf{Cr}} c$, the cost of implementing ρ on a RAM is $O((|\rho|_{\mathsf{p}} + 1) \cdot |t|)$ plus the cost of plugging and unplugging.*

OCaml implementation. In Section 7 we sketch the Pointed Crumble GLAM, a refinement of the Crumble GLAM making explicit the search for redexes and removing the need for (un)plugging, and having the *same complexity*: the cost for searching redexes and (un)plugging is negligible. More details and an implementation in OCaml of the Pointed Crumble GLAM can be found in the appendix of [6], together with the code that implements the crumbling translation. There we also discuss a parsimonious choice of data structures for the implementation of pointed environments.

6 THE OPEN CASE

6.1 The Fireball Calculus

In this section we recall the fireball calculus λ_{fire}, the simplest presentation of Open CbV, and extend it with conditionals. The extension is completely modular. For the issues of Plotkin's setting with respect to open terms and for alternative presentations of Open CbV, we refer the reader to Accattoli and Guerrieri [8, 10].

The fireball calculus was introduced without a name and studied first by Paolini and Ronchi Della Rocca [30], Ronchi Della Rocca and Paolini [32]. It has then been rediscovered by Grégoire and Leroy [20] to improve the implementation of Coq, and later by Accattoli and Sacerdoti Coen [11] to study cost models, where it was also named. We present it following Accattoli and Sacerdoti Coen [11], changing only inessential, cosmetic details.

The fireball calculus. The conditional fireball calculus $\lambda^{\mathsf{if}}_{\mathsf{fire}}$ is defined in Fig. 2. The conditional part is exactly as in the closed case. The idea is that the values of the Pif calculus are generalized to *fireballs*, by adding *inert terms*. Fireballs (noted f) and inert terms (noted i) are defined by mutual induction (in Fig. 2). For instance, x and $\lambda x.y$ are fireballs as values, while $y(\lambda x.x)$, xy, and $(z(\lambda x.x))(zz)(\lambda y.(zy))$ are fireballs as inert terms.

The main feature of inert terms is that they are open, normal, and that when plugged in a context they cannot create a redex, hence the name "inert". Essentially, they are the *neutral terms* of Open

CbV. In Grégoire and Leroy's presentation [20], inert terms are called *accumulators* and fireballs are simply called values. Variables are, morally, both values and inert terms. In Accattoli and Sacerdoti Coen [11] they were considered as inert terms, while here, for minor technical reasons we prefer to consider them as values and not as inert terms—the change is inessential.

Evaluation rules. First, CbV β-reduction is replaced by *call-by-fireball* β-reduction $\rightarrow_{\beta_f}$: the β-rule can fire, *lighting* the argument, only if the argument is a fireball (*fireball* is a catchier version of *fire-able term*). We actually distinguish two sub-rules: the usual one that *lights* values, noted $\rightarrow_{\beta_v}$, and a new one that *lights* inert terms, noted $\rightarrow_{\beta_i}$ (see Fig. 2). Second, we include all the rules about conditionals and errors, exactly as before, obtaining the evaluation relation $\rightarrow_{\mathsf{c}\beta_f}$. Note that evaluation is *weak*: it does not reduce in abstraction nor if-then-else bodies.

We endow the calculus with the (deterministic) right-to-left evaluation strategy, defined via right f-contexts R—note the production Rf, forcing the right-to-left order. A more general calculus (without conditionals) is defined in Accattoli and Guerrieri [8], for which the right-to-left strategy is shown to be complete. We omit details about the rewriting theory of the fireball calculus because our focus here is on implementations.

Example 6.1. We have $(\lambda z.z(yz))(\lambda x.x) \rightarrow_{\beta_f} (\lambda x.x)(y\,(\lambda x.x)) \rightarrow_{\beta_f} y\,(\lambda x.x)$, where the final term $y\,(\lambda x.x)$ is a fireball (and β_f-normal).

Properties. As discussed in Sect. 5, Closed CbV enjoys harmony (Prop. 2.1). The fireball calculus λ_{fire} satisfies an analogous property in the *open* setting by replacing abstractions with fireballs; we here further extend it to conditionals (Prop. 6.2.1 below). The key property of inert terms is summarized by Prop. 6.2.2: substitution of inert terms does not create or erase $\mathsf{c}\beta_f$-redexes, and hence can always be avoided. It plays a role in the design of the open abstract machine of the next section.

Proposition 6.2 (Properties of $\lambda^{\mathsf{if}}_{\mathsf{fire}}$). *Let t, u be terms.*

1. Open harmony: *t is $\mathsf{c}\beta_f$-normal if and only if t is a fireball.*
2. Inert substitutions and evaluation commute: *Let i be an inert term. Then $t \rightarrow_{\mathsf{c}\beta_f} u$ if and only if $t\{x{\leftarrow}i\} \rightarrow_{\mathsf{c}\beta_f} u\{x{\leftarrow}i\}$.*

6.2 The Open Crumble GLAM

Here we extend the Crumble GLAM defined in Sect. 5 to the case of open terms, implementing Open (Conditional) CbV, *i.e.* the conditional fireball calculus $\lambda^{\mathsf{if}}_{\mathsf{fire}}$: in this way we obtain the *Open Crumble GLAM.* The extension impacts on the core λ-calculus, while conditionals are essentially orthogonal to the issues of open terms.

Evaluated environments. First, we need to discuss the environments under which evaluation takes place. In the open case, v-crumbles and v-environments generalize to f-crumbles and f-environments, and are denoted as follows:

f-crumbles: c_f $\qquad$ f-environments: e_f

Recall that in the Crumble GLAM the already evaluated coda of the environment is made out only of practical values. Unfortunately, a syntactic characterization of f-environments (and f-crumbles) is more involved than the simple definition of v-environments.

In the Crumble GLAM, to check whether a bite b is in normal form with respect to a v-environment e_v, it suffices to check whether b is a practical value. In the open case, looking at the syntactic structure of the term is not enough: some applications are now normal, for example the bite $y\,x$ is normal with respect to the environment $e := [x{\leftarrow}I]$, but not all of them are normal, for instance $(x\,y, [x{\leftarrow}I]) \rightarrow_{\mathsf{sub}_l} (I\,y, [x{\leftarrow}I])$ as in the closed case (exact definitions are given below). Because of this additional complication, we are going to define f-environments directly in terms of their 'semantics', *i.e.* of their read-back to terms. Intuitively, fully evaluated f-environments should correspond to substitutions of fully evaluated terms in $\lambda^{\mathsf{if}}_{\mathsf{fire}}$. And since by harmony normal forms in $\lambda^{\mathsf{if}}_{\mathsf{fire}}$ are simply fireballs, it suffices to request that the read-back of every entry in a f-environment is a fireball.

Let us now define f-environments formally: e_f is a *f-environment* (resp. c_f is a *f-crumble*) if for any environment context E (resp. any crumble context C) and any crumble c such that $e_f = E\langle c\rangle$ (resp. $c_f = C\langle c\rangle$) the following two conditions hold:

1. *Read-back to fireballs:* $c_\downarrow$ is a fireball, and
2. *Unchaining practical values:* if $c_\downarrow$ is a practical value, then $c = (v, e)$ for some practical value v and some e.

The second requirement forbids v to be a variable and is crucial for capturing the correct behavior of the substitution rule $\rightarrow_{\mathsf{sub}_{var}}$, which removes the malicious chains of substitutions (of variables for variables) discussed in Sect. 5.

Transitions. The transitions of the Open Crumble GLAM:

$$((\lambda x.c)\,v, e_f) \mapsto_{\beta_f} (c \mathbin{@} [x{\leftarrow}v])^{\alpha} \mathbin{@} e_f$$

$$(\text{if true then } c \text{ else } d, e_f) \mapsto_{\mathsf{ift}} c \mathbin{@} e_f$$

$$(\text{if false then } c \text{ else } d, e_f) \mapsto_{\mathsf{iff}} d \mathbin{@} e_f$$

$$(\text{if } v \text{ then } c \text{ else } d, e_f) \mapsto_{\mathsf{ife}} (\mathsf{err}, e_f) \tag{1}$$

$$(vw, e_f) \mapsto_{\mathsf{@e}} (\mathsf{err}, e_f) \tag{2}$$

$$(x, e_f) \mapsto_{\mathsf{sub}_{var}} (e_f(x), e_f) \tag{3}$$

$$(xv, e_f) \mapsto_{\mathsf{sub}_l} (e_f(x)\,v, e_f) \tag{3}$$

$$(\text{if } x \text{ then } c \text{ else } d, e_f) \mapsto_{\mathsf{sub}_{\mathsf{if}}} (\text{if } e_f(x) \text{ then } c \text{ else } d, e_f) \tag{3}$$

(1) if $v = \lambda x.e$ or $v = \mathsf{err}$
(2) if $v \in \{\mathsf{true}, \mathsf{false}, \mathsf{err}\}$
(3) if $x \in \mathsf{dom}(e_f)$

Top level transitions are then closed by crumble contexts by setting $C\langle c\rangle \rightarrow_a C\langle d\rangle$ if $c \mapsto_a d$ for $a \in \{\beta_v, \mathsf{sub}_{var}, \mathsf{sub}_l, \mathsf{sub}_{\mathsf{if}}, \mathsf{ift}, \mathsf{iff}, \mathsf{ife}, \mathsf{@e}\}$. The transition relation $\rightarrow_{\mathsf{oCr}}$ of the Open Crumble GLAM is defined as the union of all these rules. A *principal transition* of the Open Crumble GLAM is a transition $\rightarrow_a$ for any rule $a \in \{\beta_f, \mathsf{ift}, \mathsf{iff}, \mathsf{ife}, \mathsf{@e}\}$.

There are only two differences with the transitions of the Crumble GLAM. First, $\rightarrow_{\beta_v}$ is now noted $\mapsto_{\beta_f}$ and yet it is identical to the one in the closed case (the comments about α-renaming given in Sect. 5 still hold). This is because there is a subtle difference: the argument of the β-redex may be a variable (which is a value) substituted by a inert term in the environment, thus becoming a $\rightarrow_{\beta_i}$ step (and not a $\rightarrow_{\beta_v}$ step) when read-back in $\lambda^{\mathsf{if}}_{\mathsf{fire}}$. Second, there is a slightly different side condition for the substitution transitions: it requires not only that a variable is defined in e_f (like in the closed case), but also that the corresponding term in the environment is a practical value (and not an inert term nor a variable).

Note that the substitution transitions substitute values only. The environment e_f may contain also bites that are variables or applications, but these bites are not substituted: this choice is justified by the property of $\lambda^{\mathsf{if}}_{\mathsf{fire}}$ stated in Prop. 6.2.2. Besides, avoiding the substitution of inert terms is a prerequisite for efficiency of the machine, that would otherwise be subjected to an exponential overhead due to *size explosion*, see for example Accattoli and Guerrieri [9], Accattoli and Sacerdoti Coen [11].

The harmony between evaluation rules and the syntactic definition of normal forms is witnessed by the following property.

Proposition 6.3 (Harmony for the Open Crumble GLAM). *A crumble c is* oCr*-normal if and only if it is a f-crumble.*

Example 6.4. Recall that $\overline{\delta} = (\lambda x.xx, \epsilon)$. In Example 5.2 we noted that the (open) crumble $\underline{\delta\delta(xx)}$ was stuck in the Crumble GLAM. Now instead it correctly reduces, never reaching a normal form:

$$\begin{array}{lll} \underline{\delta\delta(xx)} & = & (zw, [z{\leftarrow}\overline{\delta}\,\overline{\delta}][w{\leftarrow}xx]) \\ & \to_{\beta_v} & (zw, [z{\leftarrow}yy][y{\leftarrow}\overline{\delta}][w{\leftarrow}xx]) \\ & \to_{\mathsf{sub}_l} & (zw, [z{\leftarrow}\overline{\delta}\,y][y{\leftarrow}\overline{\delta}][w{\leftarrow}xx]) \to \cdots \end{array}$$

Implementation Theorem. The proof of the implementation theorem for the Open Crumble GLAM follows the same structure as for the Crumble GLAM in Subsect. 5.2, relying on similar but subtler invariants that can be found in the appendix of [6].

Theorem 6.5 (Implementation). *Let c be a crumble that is reachable by the Open Crumble GLAM.*

(1) Initialization: $\underline{t}_{\downarrow} = t$
(2) Principal projection: *if $c \to_a d$ then $c_{\downarrow} \to_a d_{\downarrow}$ for $a \in \{\beta_f, ift, iff, ife, @e\}$.*
(3) Overhead transparency: *if $c \to_a d$ then $c_{\downarrow} = d_{\downarrow}$ for any rule $a \in \{\mathsf{sub}_{var}, \mathsf{sub}_l, \mathsf{sub}_{\mathsf{if}}\}$.*
(4) Determinism: *the transition $\to_{\mathsf{oCr}}$ is deterministic.*
(5) Halt: *if c is $\to_{\mathsf{oCr}}$-normal then $c_{\downarrow}$ is $\to_{\mathsf{c}\beta_f}$-normal.*
(6) Overhead termination: *$\to_a$ terminates, for any rule $a \in \{\mathsf{sub}_{var}, \mathsf{sub}_l, \mathsf{sub}_{\mathsf{if}}\}$.*

Therefore, the Open Crumble GLAM, the right-to-left conditional fireball evaluation $\to_{\mathsf{c}\beta_f}$ and the read-back $(\cdot)_{\downarrow}$ form an implementation system.

Complexity. The complexity analysis is identical to the one in Subsect. 5.3. Indeed, once the search for the next redex and (un)plugging are neglected, the two machines only differ by the additional $O(1)$ side condition for the substitution transitions.

Theorem 6.6 (The Open Crumble GLAM is bilinear up to search and (un)plugging). *For any term t and any Open Crumble GLAM execution $d \colon \underline{t} \to^*_{\mathsf{oCr}} c$, the cost of implementing ρ on a RAM is $O((|\rho|_{\mathsf{p}} + 1) \cdot |t|)$ plus the cost of plugging and unplugging.*

OCaml implementation. Following the same pattern of the closed case, in Section 7 we introduce a machine making explicit the search for redexes and removing the need of (un)plugging, so as to show that their cost is negligible. The OCaml code implementing this further machine is in the appendix of [6], along with a detailed discussion of the adopted data structures. The code for the open and closed machines is identical but for five lines: three implement the additional check for practical values in the substitution transitions, the others consider also inert terms in the search transition.

7 THE (OPEN) POINTED CRUMBLE GLAM

In the abstract machines considered so far the *search for the next redex* is implicit in definition of the evaluation rules, as it corresponds to applying rules to crumbles with already evaluated environments and to factoring out the crumble context.

Here we sketch how to make the search explicit by introducing a variant of the Crumble GLAM called *Pointed Crumble GLAM*. The new search transitions have constant cost and the machine is bilinear—the detailed study is in the appendix of [6].

Pointed crumbles and pointed environments. The key idea behind the Pointed Crumble GLAM is to avoid (un)plugging in the rules by letting them act on *pointed crumbles*, *i.e.* on crumbles where a *pointer* marks explicitly the dividing point between the evaluated coda and the crumbled term of the currently active crumble. A pointed crumble $(b, e[x{\leftarrow}b'] ¦ e_v)$ represents the crumble $C\langle(b', e_v)\rangle$, where $C = (b, e[x{\leftarrow}\langle\cdot\rangle])$ is the crumble context, (b', e_v) is the active crumble, and e_v is the evaluated coda. If (b', e_v) is a Crumble GLAM a-redex (for rule $a \in \{\beta_v, \mathsf{sub}_{var}, \mathsf{sub}_l\}$), the Pointed Crumble GLAM shall reduce according to the corresponding a-transition that also takes care of setting (in $O(1)$) the pointer to the rightmost unevaluated crumble. Otherwise, by harmony (Prop. 5.3), b' must be a crumbled value v and therefore the pointer is moved (in $O(1)$) one step to the left, looking for the next redex, via the search transition $(b, e[x{\leftarrow}v] ¦ e_v) \to_{\mathsf{sea}} (b, e ¦ [x{\leftarrow}v]e_v)$.

Unfortunately, there is an annoying technical issue. Not all pointed configurations are of the form $(b, [x{\leftarrow}b'] ¦ e_v)$: the configurations $(b, ¦\, e_v)$ must be also taken into account and reduced if b is not a crumbled value. However, there is no simple way to describe transitions that act uniformly on configurations $(b, ¦\, e_v)$ and $(b, e[x{\leftarrow}b'] ¦ e_v)$ without duplicating the rules or without reintroducing a notion of contextual closure. To solve the issue, we abandon pointed crumbles and adopt pointed environments instead.

A *pointed environment* $([x{\leftarrow}b]e ¦ e_v)$ is just a representation of a pointed crumble $(b, e ¦ e_v)$. The leftmost variable x in a pointed environment can be understood as the name given to the machine output. It plays a role similar to the outermost λ-abstraction introduced by CPS translations, that binds the continuation that is fed with the output of the evaluation. In particular, a normal pointed environment $(¦\, [x{\leftarrow}v]e_v)$ represents the normal crumble (v, e_v).

Formal definition of pointed environments and transitions. Pointed environments are defined as $e_{¦} \coloneqq e ¦ e'$ for any non-pointed environments e and e' where at least one among e or e' is non-empty. The *translation* $\iota(\cdot)$ embeds crumbles into pointed environments: $\iota(b, e) \coloneqq [x{\leftarrow}b]e¦\epsilon$, where x is any variable fresh in b and e.

The transition rules of the Pointed Crumble GLAM are:

$$\begin{array}{rcll} e[x{\leftarrow}(\lambda y.c)\, v] ¦ e_v & \to_{\beta_v} & e[x{\leftarrow}b]e'[z{\leftarrow}v] ¦ e_v & (i) \\ e[y{\leftarrow}x] ¦ e_v & \to_{\mathsf{sub}_{var}} & e[y{\leftarrow}e_v(x)] ¦ e_v & (ii) \\ e[y{\leftarrow}xv] ¦ e_v & \to_{\mathsf{sub}_l} & e[y{\leftarrow}e_v(x)v] ¦ e_v & (ii) \\ e[x{\leftarrow}b] ¦ e_v & \to_{\mathsf{sea}} & e ¦ [x{\leftarrow}b]e_v & (iii) \end{array}$$

where (we omitted the rules for conditionals, for the sake of brevity)

i. $\lambda z.(b, e') := (\lambda y.c)^{\alpha}$ with $(e[x{\leftarrow}b]e'[z{\leftarrow}v] \mid e_v)$ well-named;
ii. if $x \in \mathsf{dom}(e_v)$;
iii. if none of the other rules is applicable, *i.e.* when b is an abstraction or when b is x or xv but x is not defined in e_v.

In the appendix of [6], we prove that the Pointed Crumble GLAM simulates the Crumble GLAM following exactly the same schema already used in the paper, namely they form an implementation system. We also provide the complexity analysis, and smoothly lift everything to the open case, by studying the *Open Pointed Crumble GLAM* (which simulates the Open Crumble GLAM and whose transition function is noted $\to_{\mathsf{poCr}}$). We obtain the following result, that sums up the study in this paper (Point 2 is a corollary of Point 1).

Theorem 7.1 (The Open (Pointed) Crumble GLAM is bilinear). *Let t be a term.*

(1) for any Open Pointed Crumble GLAM execution $\rho\colon \iota(\underline{t}) \to^{*}_{\mathsf{poCr}} e_{\mid}$, the cost of implementing ρ on a RAM is $O((|\rho|_{\mathsf{p}} + 1) \cdot |t|)$.
(2) for any Open Crumble GLAM execution $\rho\colon \underline{t} \to^{*}_{\mathsf{oCr}} c$, the cost of implementing ρ on a RAM is $O((|\rho|_{\mathsf{p}} + 1) \cdot |t|)$.

8 EXTENSIONS

Left-to-right CbV. The (right-to-left) Crumble GLAM can also implement a left-to-right strategy for the Pif calculus. The only change concerns the crumbling transformation, that on applications has to put the environment coming from the (transformation of the) left subterm on the right of the one coming from the right subterm.

Call-by-need. The crumbling technique applies also to call-by-need machines. There are however a few differences. First, the machine does no longer explore sequentially the environment from right-to-left, it rather starts on the left and then jumps back and forth, by need. Then the definition of evaluation contexts is trickier, especially in the open case.

Strong CbV. Simply designing an abstract machine for strong reduction is relatively easy. However the easy machines are not *bilinear*, and not even polynomial.

The needed optimizations to make them reasonable (i.e. polynomial or bilinear) are clear, they are the same at work in the open case (or in the call-by-name case):

(1) substitute only abstraction and not inert terms, and
(2) do not substitute abstractions on variable occurrences that are not applied.

These principles however have different consequences in different settings. In particular, (2) implies that some abstraction are kept shared forever, and a *strong* CbV approach has to evaluate them (while the open setting does not) and only once, thus it has to evaluate them while they are shared, adding a call-by-need flavor.

There are two difficulties. First, the specification of the search for redexes, that becomes involved and requires many machine transitions—the crumbling technique is meant to help here. Second, the proof of correctness of the machine.

All proofs of correctness in the literature (including those in this paper) are simulations up to sharing based on a *bijection* of β-redexes (or principal steps) between the abstract machine and the λ-calculus strategy (one half of the bijection is the principal projection property of implementation systems in Section 5.2, the other half is implied by the other properties).

The evaluation under *shared* abstraction required by CbV strong evaluation breaks the usual bijection of β-redexes (as one β-transition of the machine is mapped to *many* β-steps on the calculus, and not necessarily those of a standard strategy), thus forbidding to employ the standard technique for proving correctness.

The new proof technique for correctness for reasonable strong CbV and the intricacies of the search for redexes in the strong case, do deserve to be studied carefully, and are thus left to future work.

9 COMMENTS ON RELATED WORKS

Here we discuss Kennedy's potential slowdown and provide a counterexample to the scalability of the CPS translation to open terms.

Kennedy. Kennedy [24] compares three different calculi: a monadic calculus, which has ES, a calculus of administrative normal forms (ANFs) and the image of a CPS transformation. In the monadic calculus β_v-redexes can be hidden by ES which need to be commuted to reveal the β_v-redex. Kennedy shows an example (see Fig. 3) where the number of commutations is not bounded linearly by the number of β_v-steps and blames the inefficiency of his compiler on that. In his example, the number of commutations is *quadratic* in the number of β_v-steps, since the i^{th} β_v-step is immediately followed by i commutation steps.

ANFs are just canonical shapes of monadic terms where the topmost term and the body of each abstraction is a crumble, *i.e.* a term together with a list of ES that map variables to terms (instead of crumbles). Kennedy rightly observes that ANFs are not preserved by standard β_v-reduction, and thus, after each β_v-step, some commutative steps are required to reach the ANF shape. Kennedy too hastily concludes that the quadratic blowup also affects the ANF calculus, since its quadratic example stands in the ANF fragment.

However, Kennedy misses the fact that the ES in ANFs form a list and that the commutations steps altogether just implement the append of two lists. Since append can be implemented in constant time, the complexity of evaluation in the ANF calculus is just linear (and not quadratic) in the number of β_v-steps. This is the same complexity we achieved for the Crumble and Open Crumble GLAM.

Danvy and Filinski. In Danvy and Filinski [17] the CPS transformation is shown to scale up to open terms (their Theorem 2). On open terms, however, they consider Plotkin's CbV operational semantics λ_{Plot}, which is *not adequate* (it is adequate only for closed terms, see Accattoli and Guerrieri [8, 10] and Guerrieri [21]). When one considers one of the equivalent adequate CbV semantics in Accattoli and Guerrieri [8, 10] for the open case, for instance the fireball calculus λ_{fire}, then the properties of the CPS no longer hold, in particular it does not commute with evaluation, as the following example shows. Take the following open term $t := (\lambda x.\lambda y.y)(zz)v$, where v is a value, say a distinguished variable. In λ_{Plot} the term t is β_v-normal, but in λ_{fire} we have:

$$t := (\lambda x.\lambda y.y)(zz)v \to_{\beta_f} (\lambda y.y)v \to_{\beta_f} v$$

Now, consider the CPS translation $\mathsf{cps}(t)$ of t, according to the definition in Danvy and Filinski [17]. We use λ for standard ("dynamic", in Danvy's terminology) abstraction, and Λ and @ for "static" abstraction and "static" prefix application, respectively. If a

$$
\begin{array}{l}
t := (z_1 x_0)[z_1 \leftarrow \lambda x_1 . b y_1 [y_1 \leftarrow z_2 x_1]][z_2 \leftarrow \lambda x_2 . b y_2 [y_2 \leftarrow z_3 x_2]] \ldots [z_n \leftarrow \lambda x_n . b y_n [y_n \leftarrow b x_n]] \\
(\rightarrow_{\beta_v} \rightarrow_{\text{let}})\ (\rightarrow_{\beta_v} \rightarrow_{\text{let}}^2) \cdots (\rightarrow_{\beta_v} \rightarrow_{\text{let}}^i) \\
\quad (z_1 x_0)[z_1 \leftarrow \lambda x_1 . b y_1 [y_1 \leftarrow z_2 x_1]][z_2 \leftarrow \lambda x_2 . b y_2 [y_2 \leftarrow z_3 x_2]] \ldots \\
\quad \ldots [z_{n-i-1} \leftarrow \lambda x_{n-i-1} . b y_{n-i-1} [y_{n-i-1} \leftarrow z_{n-i} x_{n-i-1}]][z_{n-i} \leftarrow \lambda x_{n-i} . b y_{n-i} [y_{n-i} \leftarrow b y_{n-i+1}][y_{n-i+1} \leftarrow b y_{n-i+2}] \ldots [y_n \leftarrow b x_{n-i}]] \\
\rightarrow_{\beta_v} \\
\quad (z_1 x_0)[z_1 \leftarrow \lambda x_1 . b y_1 [y_1 \leftarrow z_2 x_1]][z_2 \leftarrow \lambda x_2 . b y_2 [y_2 \leftarrow z_3 x_2]] \ldots \\
\quad \ldots [z_{n-i-1} \leftarrow \lambda x_{n-i-1} . b y_{n-i-1} [y_{n-i-1} \leftarrow b y_{n-i} [y_{n-i} \leftarrow b y_{n-i+1}][y_{n-i+1} \leftarrow b y_{n-i+2}] \ldots [y_n \leftarrow b x_{n-i-1}]]] \\
\rightarrow_{\text{let}}^{i+1} \\
\quad (z_1 x_0)[z_1 \leftarrow \lambda x_1 . b y_1 [y_1 \leftarrow z_2 x_1]][z_2 \leftarrow \lambda x_2 . b y_2 [y_2 \leftarrow z_3 x_2]] \ldots \\
\quad \ldots [z_{n-i-1} \leftarrow \lambda x_{n-i-1} . b y_{n-i-1} [y_{n-i-1} \leftarrow b y_{n-i}][y_{n-i} \leftarrow b y_{n-i+1}] \ldots [y_n \leftarrow b x_{n-i-1}]]
\end{array}
$$

Figure 3: Kennedy's example of evaluation in the monadic calculus where the number of commutation steps is quadratic in the number of β_v-steps ($\rightarrow^i$ stands for the composition of i $\rightarrow$-steps). The i^{th} β_v-step (which can reduce under abstractions) is immediately followed by i commutation steps $\rightarrow_{\text{let}}$ that just append two lists of substitutions moving one substitution at a time. Thus, to reach a normal form one needs n β_v-steps and $n(n+1)/2$ let-steps. In the Crumble and Open Crumble GLAM instead, the commutation steps are integrated in the β_v-rule simply by appending the two lists in constant time.

generalized version of Theorem 2 in Danvy and Filinski [17] held in the open case, one would expect that $@(\mathsf{cps}(t))I$ (where $I := \lambda z.z$) evaluates to v, as v is a value. But, even using an unrestricted β-reduction that goes under abstraction as evaluation, we obtain (we reduce all static redexes first, followed by all dynamic redexes):

$$
\begin{array}{ll}
& @(\mathsf{cps}(t))I \\
= & (\Lambda k.@(\Lambda x.@(\Lambda y.@y(\lambda w.\lambda a.@(\Lambda b.@b(\lambda c.\lambda d.@(\Lambda e.@ec)(\Lambda e.de))) \\
& (\Lambda b.ab)))(\Lambda y.@(\Lambda j.@(\Lambda a.@az)(\Lambda a.@(\Lambda b.@bz)(\Lambda b.(ab)(\lambda c.@jc)))) \\
& (\Lambda w.(yw)(\lambda a.@xa))))(\Lambda x.@(\Lambda y.@yv)(\Lambda y.(xy)(\lambda w.@Kw))))I \\
\rightarrow_\beta^* & (zz)(\lambda x.((\lambda y.\lambda w.w(\lambda a.\lambda b.ba))x)(\lambda y.yv(\lambda w.Iw))) \\
\rightarrow_\beta^* & (zz)(\lambda x.v)
\end{array}
$$

where $(zz)(\lambda x.v)$ is not even β-equivalent to v. The CPS translation—like Plotkin's calculus—gets stuck trying to evaluate zz, whereas the term reduces to v in the fireball calculus.

Summing up, we are not claiming that Theorem 2 in Danvy and Filinski [17] is false, but just that it does not mean that their CPS transformation scales up to open terms: to prove scalability, one should use an adequate CbV evaluation for open terms (such as the one of the fireball calculus), instead of Plotkin's one. Our counter-example shows that Danvy's and Filinski's CPS does not scale up to open terms with an adequate CbV operational semantics for them.

This problem affects also other CPS translations, such as the ones defined by Plotkin [31] or by Lassen [26]. Likely, this is the reason why Lassen [26] states his Theorem 4.6 (the analogous of Theorem 2 in Danvy and Filinski [17]) only for closed terms.

10 CONCLUSIONS

This paper studies abstract machines working on crumbled forms with respect to design, efficiency, scalability, and implementations, putting emphasis on the role played by environments and providing a detailed technical development. In particular, we study the crumble setting on top of global environments—in future work we would like to explore the more technical case of local environments.

At the level of design, switching to crumbled forms removes the need for machine data structures such as the applicative stack or the dump, as they are encoded in crumbled environments.

At the level of efficiency, the evaluation of crumbled forms does not require any overhead: crumble abstract machines are linear in the number of steps of the calculus and in the size of the initial term, exactly as ordinary abstract machines with global environments.

At the level of scalability, everything—including the complexity—smoothly scales up from the closed case, relevant for programming languages, to the more delicate case of open terms, needed to implement proof assistants. As shown in Section 9, CPS translations do not smoothly scale up to the open case (contrary to what claimed by Danvy and Filinski [17]), so that our work shows an advantage of the crumbling transformation in this setting.

At the level of implementations, we stress the different operations on crumbled environments (sequential access and concatenation) and provide a concrete implementation, which does not suffer from the potential slowdown of crumbled forms pointed out by Kennedy [24] (see Section 9).

In future work we plan to apply our results to the design of abstract machines for strong call-by-value and call-by-need evaluation. Preliminary results suggest that the simplification to the code noticed in the open case is preserved and even amplified in the harder case of strong evaluation.

Acknowledgments. This work has been partially funded by the ANR JCJC grant COCA HOLA (ANR-16-CE40-004-01) and by the EPSRC grant EP/R029121/1 "Typed Lambda-Calculi with Sharing and Unsharing".

REFERENCES

[1] Beniamino Accattoli. 2012. An Abstract Factorization Theorem for Explicit Substitutions. In *23rd International Conference on Rewriting Techniques and Applications, RTA 2012 (LIPIcs)*, Vol. 15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 6–21. https://doi.org/10.4230/LIPIcs.RTA.2012.6

[2] Beniamino Accattoli. 2016. The Useful MAM, a Reasonable Implementation of the Strong λ-Calculus. In *Logic, Language, Information, and Computation -*

23rd International Workshop, WoLLIC 2016 (Lecture Notes in Computer Science), Vol. 9803. Springer, 1–21. https://doi.org/10.1007/978-3-662-52921-8_1

[3] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2014. Distilling abstract machines. In *19th ACM SIGPLAN International Conference on Functional Programming, ICFP 2014*. ACM, 363–376. https://doi.org/10.1145/2628136.2628154

[4] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2015. A Strong Distillery. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015 (Lecture Notes in Computer Science)*, Vol. 9458. Springer, 231–250. https://doi.org/10.1007/978-3-319-26529-2_13

[5] Beniamino Accattoli and Bruno Barras. 2017. Environments and the Complexity of Abstract Machines. In *19th International Symposium on Principles and Practice of Declarative Programming, PPDP 2017*. ACM, 4–16. https://doi.org/10.1145/3131851.3131855

[6] Beniamino Accattoli, Andrea Condoluci, Giulio Guerrieri, and Claudio Sacerdoti Coen. 2019. Crumbling Abstract Machines (Extended Version). *CoRR* abs/1907.06057 (2019).

[7] Beniamino Accattoli and Ugo Dal Lago. 2016. (Leftmost-Outermost) Beta Reduction is Invariant, Indeed. *Logical Methods in Computer Science* 12, 1 (2016). https://doi.org/10.2168/LMCS-12(1:4)2016

[8] Beniamino Accattoli and Giulio Guerrieri. 2016. Open Call-by-Value. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016 (Lecture Notes in Computer Science)*, Vol. 10017. Springer, 206–226. https://doi.org/10.1007/978-3-319-47958-3_12

[9] Beniamino Accattoli and Giulio Guerrieri. 2017. Implementing Open Call-by-Value. In *Fundamentals of Software Engineering - 7th International Conference, FSEN 2017 (Lecture Notes in Computer Science)*, Vol. 10522. Springer, 1–19. https://doi.org/10.1007/978-3-319-68972-2_1

[10] Beniamino Accattoli and Giulio Guerrieri. 2018. Types of Fireballs. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018 (Lecture Notes in Computer Science)*, Vol. 11275. Springer, 45–66. https://doi.org/10.1007/978-3-030-02768-1_3

[11] Beniamino Accattoli and Claudio Sacerdoti Coen. 2015. On the Relative Usefulness of Fireballs. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015*. IEEE Computer Society, 141–155. https://doi.org/10.1109/LICS.2015.23

[12] Beniamino Accattoli and Claudio Sacerdoti Coen. 2017. On the value of variables. *Information and Computation* 255 (2017), 224–242. https://doi.org/10.1016/j.ic.2017.01.003

[13] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. 1995. The Call-by-Need Lambda Calculus. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*. ACM Press, 233–246. https://doi.org/10.1145/199448.199507

[14] Pierre Crégut. 1990. An Abstract Machine for Lambda-Terms Normalization. In *LISP and Functional Programming*. 333–340. https://doi.org/10.1145/91556.91681

[15] Olivier Danvy. 1994. Back to Direct Style. *Science of Computer Programming* 22, 3 (1994), 183–195. https://doi.org/10.1016/0167-6423(94)00003-4

[16] Olivier Danvy. 2003. A New One-Pass Transformation into Monadic Normal Form. In *Compiler Construction, 12th International Conference, CC 2003 (Lecture Notes in Computer Science)*, Vol. 2622. Springer, 77–89. https://doi.org/10.1007/3-540-36579-6_6

[17] Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science* 2, 4 (1992), 361–391. https://doi.org/10.1017/S0960129500001535

[18] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations (with retrospective). In *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection, PLDI 1993*. ACM, 502–514. https://doi.org/10.1145/989393.989443

[19] Álvaro García-Pérez, Pablo Nogueira, and Juan José Moreno-Navarro. 2013. Deriving the full-reducing Krivine machine from the small-step operational semantics of normal order. In *15th International Symposium on Principles and Practice of Declarative Programming, PPDP'13*. ACM, 85–96. https://doi.org/10.1145/2505879.2505887

[20] Benjamin Grégoire and Xavier Leroy. 2002. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ICFP '02*. ACM, 235–246. https://doi.org/10.1145/581478.581501

[21] Giulio Guerrieri. 2019. Towards a Semantic Measure of the Execution Time in Call-by-Value lambda-Calculus. In *Proceedings Twelfth Workshop on Developments in Computational Models and Ninth Workshop on Intersection Types and Related Systems, DCM/ITRS 2018. (EPTCS)*, Vol. 293. 57–72. https://doi.org/10.4204/EPTCS.293.5

[22] Giulio Guerrieri, Luca Paolini, and Simona Ronchi Della Rocca. 2017. Standardization and Conservativity of a Refined Call-by-Value lambda-Calculus. *Logical Methods in Computer Science* 13, 4 (2017). https://doi.org/10.23638/LMCS-13(4:29)2017

[23] John Hatcliff and Olivier Danvy. 1994. A Generic Account of Continuation-Passing Styles. In *21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1994*. ACM Press, 458–471. https://doi.org/10.1145/174675.178053

[24] Andrew Kennedy. 2007. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*. ACM, 177–190. https://doi.org/10.1145/1291151.1291179

[25] Arne Kutzner and Manfred Schmidt-Schauß. 1998. A Non-Deterministic Call-by-Need Lambda Calculus. In *Third ACM SIGPLAN International Conference on Functional Programming, ICFP 1998*. ACM, 324–335. https://doi.org/10.1145/289423.289462

[26] Søren B. Lassen. 2005. Eager Normal Form Bisimulation. In *20th IEEE Symposium on Logic in Computer Scienc, LICS 2005*. IEEE Computer Society, 345–354. https://doi.org/10.1109/LICS.2005.15

[27] John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1993*. ACM Press, 144–154. https://doi.org/10.1145/158511.158618

[28] John Maraist, Martin Odersky, and Philip Wadler. 1998. The Call-by-Need Lambda Calculus. *Journal of Functional Programming* 8, 3 (1998), 275–317.

[29] Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[30] Luca Paolini and Simona Ronchi Della Rocca. 1999. Call-by-value Solvability. *ITA* 33, 6 (1999), 507–534. https://doi.org/10.1051/ita:1999130

[31] Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theoretical Computer Science* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

[32] Simona Ronchi Della Rocca and Luca Paolini. 2004. *The Parametric λ-Calculus – A Metamodel for Computation*. Springer. https://doi.org/10.1007/978-3-662-10394-4

[33] Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation* 6, 3-4 (1993), 289–360.

[34] Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. *Journal of Functional Programming* 7, 3 (1997), 231–264.

[35] Christopher P. Wadsworth. 1971. *Semantics and pragmatics of the lambda-calculus*. PhD Thesis. Oxford. Chapter 4.