Strong Call-by-Value is Reasonable, Implosively

Strong Call-by-Value is Reasonable, Implosively

Beniamino Accattoli
Inria & LIX, École Polytechnique

Andrea Condoluci
Tweag I/O

Claudio Sacerdoti Coen
University of Bologna

***Abstract*—Whether the number of β-steps in the λ-calculus can be taken as a reasonable time cost model (that is, polynomially related to the one of Turing machines) is a delicate problem, which depends on the notion of evaluation strategy. Since the nineties, it is known that weak (that is, out of abstractions) call-by-value evaluation is a reasonable strategy while Lévy's optimal parallel strategy, which is strong (that is, it reduces everywhere), is not. The strong case turned out to be subtler than the weak one. In 2014 Accattoli and Dal Lago have shown that strong call-by-name is reasonable, by introducing a new form of useful sharing and, later, an abstract machine with an overhead quadratic in the number of β-steps.**

Here we show that also strong call-by-value evaluation is reasonable for time, via a new abstract machine realizing useful sharing and having a linear overhead. Moreover, our machine uses a new mix of sharing techniques, adding on top of useful sharing a form of implosive sharing, which on some terms brings an exponential speed-up. We give examples of families that the machine executes in time *logarithmic* in the number of β-steps.

I. Introduction

In the last few years, the understanding of the time cost models of the λ-calculus has attracted considerable attention. The beauty of the λ-calculus is that it is an abstract for-malism, distant from low-level implementation details, while still achieving the same expressive power as Turing machines: it suffices a s ingle β -rule, b ased o n a n atural n otion of substitution. This is however also its main drawback, as the substitution it is based upon is a non-atomic operation that may duplicate whole sub-programs. A natural question then is how to measure the time of programs expressed in the λ-calculus. Of course, one wants a *reasonable* cost model in the sense of Slot and van Emde Boas [55], that is, preserving the notion of polynomial time complexity as defined o n Turing machines.

The candidate measure for time is the number of β-steps to normal form. At first sight, this approach does not seem to work. A first issue is that one has to be more precise because there are many different evaluation strategies and notions of normal form in the λ-calculus. There is however a second bigger issue, called *size explosion*, that affects every evaluation strategy. Namely, there are families $\{t_n\}_{n\in\mathbb{N}}$ of λ-terms such that t_n produces in n β-steps—independently of the strategy—a result r_n of size exponential in n. Then the chosen measure of time—namely n—does not even account for the time to write down the result, whose size is exponential in n.

How to Stop Worrying and Love the Bomb: The way out of this apparent *cul-de-sac* is to turn to evaluation *up to sharing*, where sharing is used to provide compact representations of results, avoiding the explosion. It then turns out that (for some natural strategies and notions of normal form) the number of β-steps is a reasonable time cost model. The point is subtle, let us be precise.

The idea is to first fix a strategy $\to_{\mathsf{s}}$ (together with its notion of normal form) in the λ-calculus, which is kept as a specification, reference system. Then, to study $\to_{\mathsf{s}}$ via a refined λ-calculus with sharing—think of an abstract machine—showing that $\to_{\mathsf{s}}$ can be implemented with an overhead polynomial in the number of $\to_{\mathsf{s}}$ steps, producing as output a term with sharing. The exponential explosion is then moved to the process of *unsharing* output terms. Luckily, unsharing can essentially always be avoided (unless one really needs to print the unshared output) as terms with sharing can be manipulated efficiently without having to unshare them, see Condoluci, Accattoli, and Sacerdoti Coen [28].

Subterm Sharing and Closed Evaluation: Sharing is an overloaded word, indicating a number of very different techniques in the literature about decompositions of the λ-calculus. The most basic one can be deemed *subterm sharing*—itself coming in a number of variants—that amounts to annotate terms with delayed substitutions, coming from β-steps that have been encountered during the evaluation process. Such annotations may take the form of let-expressions, explicit substitutions, or environments in abstract machines. Subterm sharing is enough to show that the number of β-steps is a reasonable cost model for both weak call-by-name (shortened to CbN) and weak call-by-value (CbV) evaluation (*weak* = out of abstractions) with closed terms. These two settings are here referred to as Closed CbN and Closed CbV. The latter models evaluation in CbV functional programming languages such as OCaml, and the fact that it is reasonable has been the first result in the literature about reasonable strategies for the λ-calculus, due to Blelloch and Greiner [26]. Similar results have also been obtained by Sands, Gustavsson, and Moran [54] and Dal Lago and Martini [33], [32], [31].

Useful Sharing and Strong (CbN) Evaluation: Subterm sharing is not enough beyond the closed case, that is, when evaluation may take place under abstraction and terms may be open—what we refer to as the *strong λ-calculus*. Namely, there are exploding families whose strong evaluation with subterm sharing takes exponential time, independently of the evaluation strategy. For some time, indeed, it has been an open question whether there are strong strategies that can be implemented within a reasonable overhead. The community used to believe that it was not the case, because of Asperti

and Mairson's result that the Lévy's optimal (strong) strategy is not reasonable [18].

The question was settled by Accattoli and Dal Lago, showing that Strong CbN is reasonable: the number of call-by-name leftmost(-outermost) evaluation β-steps, which is a strong strategy, is a reasonable time cost model [7]. For proving their result, they introduce the new layer of *useful sharing*, operating on top of subterm sharing, and show that this is mandatory. Useful sharing amounts to do minimal unsharing work, namely only when it contributes to create β-steps, while avoiding to unfold the sharing when it only makes the term grow in size.

In [7], the authors prove a polynomial overhead without investigating the degree. Later on, Accattoli provided an abstract machine—the only reasonable strong machine in the literature—with quadratic overhead when implemented on random access machines (RAM) [3].

Knowing that leftmost evaluation (sometimes referred to as *normal order*) is reasonable is theoretically valuable. Because it answers an important questions, but also because leftmost evaluation is a sort of canonical strategy for the strong λ-calculus. At the same time, however, it is not of much practical value because leftmost evaluation might be inefficient, and Strong CbV or Strong Call-by-Need are preferred in practice. For instance, both are used in the implementation of Coq.

A. Contributions of the Paper

We prove, for the first time, that also Strong CbV is reasonable for time. The calculus for Strong CbV that we adopt is Accattoli and Paolini's *value substitution calculus* [11] (shortened to VSC), for which we consider an *external strategy* playing the same role played by the leftmost strategy in Strong CbN (proved normalizing by Accattoli et al. in [10]).

The main contribution is a new abstract machine, the *strong crumbling abstract machine* (SCAM), that implements the external strategy of the VSC (Theorem X.4) and that via subterm and useful sharing does so within a *bilinear* overhead (Thm. XI.6), that is, linear in the number of β-steps and in the size of the initial term, when implemented on RAM, improving over Accattoli's quadratic bound. The SCAM actually goes considerably further, adding a form of *implosive sharing* (surveyed below) which on some terms brings an exponential speed-up, evaluating them in time *logarithmic* (!) in the number of β-steps, as we show on an example (Prop. XII.1). Since implosive sharing forbids to apply the usual proof technique for the correctness of abstract machines, we develop a new more flexible one, based on the notion of *relaxed implementation*, in Sect. V. Last, we provide a prototype implementation of our machine in OCaml.

B. Motivations

First and foremost, our motivation is foundational. We want to contribute to the study of reasonable cost models, showing that Strong CbV is reasonable for time. We also strive to obtain the best bounds for implementing strong evaluations because, after decades of research, how to best implement (strong) β-reduction is still an open problem.

Another motivation comes from the theory of proof assistants, where strong evaluation plays a role. Typically, settings such as Coq or Agda use strong evaluation to implement the β-conversion test, used for type checking with dependent types. In particular, one of the abstract machines at work in Coq, due to Grégoire and Leroy [41], relies on call-by-value.

Bounds for β-Conversion: The *pure algorithm* for testing β-conversion of two terms t and u first reduces them to normal form and then tests the results for equality. In general, conversion is undecidable, because t or u may diverge, but one may ask—when t and u are normalizable—what is the complexity of checking conversion. Without sharing, the pure algorithm is clearly exponential. By combining our results with the linear time algorithm for equality up to sharing by Condoluci, Accattoli, and Sacerdoti Coen [28], we obtain a pure call-by-value algorithm using sharing, and working in time linear in the number of (CbV) β-steps and in the size of the initial terms. We are not aware of other similar bounds in the literature, nor of any algorithmic study of β-conversion.

Beware: even though this work provides foundations for the implementation of proof assistants, we do not aim at direct applications. This is because proof assistants do not usually implement conversion via the pure algorithm, as they rest on a number of heuristics to shortcut it, see Sacerdoti Coen [53].

C. Implosive Sharing and All That

Once subterm sharing is adopted, it is possible to also *evaluate* inside shared subterms, thus sharing *evaluations*, not just subterms. The consequence is that one β-step in the shared settings maps to potentially *many* β-steps in the λ-calculus, creating in some cases a *steps explosion*, or, dually, an implosion: n β-steps in the λ-calculus may in some cases *implode* up to $\log n$ steps in the refinement with sharing.

The terminology *implosive sharing* is ours but not the (previously nameless) concept: the literature contains implosive evaluation strategies, such as Wadsworth's call-by-need [56] (shortened to CbNeed) or Lévy's optimal reduction [45].

Implosive sharing poses two technical issues. First, proving correctness, because one needs to relate a single β-step in the sharing setting with potentially many β-steps in the unshared one, which is always involved. The second challenge is complexity analyses, because implosive sharing at times breaks the so-called *subterm invariant*: the key property that duplicated terms along the whole evaluation with sharing are subterms of the initial term, which is essential for complexity analyses, and it is used in all existing proofs that a strategy is reasonable.

Mixing Implosive and Useful Sharing: There is a degree of freedom in the design of useful sharing. Accattoli and Dal Lago use a non-implosive approach which is naturally suggested by the CbN setting that they study. We adopt here an alternative implosive approach, naturally suggested by the CbV setting. This and other design choices of our SCAM, such as garbage collection and the light form of compilation called *crumbling*, are detailed in Sect. VI.

Because of implosive sharing, correctness is the most demanding theorem of the paper, for which we develop a new

abstract approach, deemed *relaxed implementation*, that we then apply concretely. They key idea is modeling the one-to-many phenomenon induced by implosive sharing via a *parallel* strategy on the calculus. The complexity analysis of the SCAM, instead, is a smooth adaptation of others in the literature, because the implosive sharing of the SCAM is carefully designed as to not clash with the subterm invariant.

Space: As most environment machines in the literature, the SCAM uses space linearly in its time consumption, and it is then space inefficient. In contrast to the literature however, the SCAM does implement garbage collection, and we strived to make our prototype implementation parsimonious in space.

We do not address the study of a reasonable space cost model—the existence of one for the λ-calculus is an open problem. There is a recent partial result by Forster, Kunze, and Roth [39], but their space cost model—namely, the size of the term—can only measure linear and super-linear space. In order to study relevant space complexity classes such as L, one needs to be able to measure *sub-linear* space, and thus the result in [39] is not a solution for the general problem.

D. Related Work

About Abstract Machines: The study of machines for strong evaluation is a blind spot of the field, despite the relevance for the implementation of proof assistants. There are very few strong machines in the literature. The ones by Crégut [29], [40] (CbN), Biernacka et al. [22] (CbV), and Biernacka and Charatonik [23] (CbNeed) all have exponential overhead. The last two works are based on Ager et al. functional correspondence [14] and Danvy and Nielsen (generalized) refocusing [34], [25]. After submitting our work, we became aware of an independent, concurrent, and currently unpublished work by Biernacka et al. also proving that Strong CbV is reasonable for time [24], and using a different approach. De Carvalho [36] and Ehrhard and Regnier [37] study variants of Crégut's machine for denotational purposes.

Coq uses more than one abstract machine for strong evaluation. The CbV one due to Grégoire and Leroy [41] is—perhaps surprisingly—not really a strong machine. It is obtained by iterating under abstractions a CbV machine for weak evaluation with open terms—a setting sometimes called *Open CbV* [8]. Their machine for Open CbV has exponential overhead (if implemented as defined in [41]), and the iteration is also naïve and costly (namely it unfolds sharing before iterating, thus potentially *exploding* in size), adding a further exponential cost. Iterating an open machine is actually very subtle: Accattoli and Guerrieri in [9] show that, even without sharing unfolding, and even when the open machine is reasonable, iterating may not give a reasonable machine for Strong CbV, as the iteration may introduce an exponential blow up. Abstract machines for Open CbV have then been studied in-depth by Accattoli and co-authors in [12], [9], [6], and optimized as to be reasonable and with linear overhead, but never extended to Strong CbV.

Coq also uses a strong machine performing CbNeed evaluation, designed and studied by Barras' in his PhD thesis [21], and for which no complexity results are known—its correctness to our knowledge has never been fully proved.

About Implosive Sharing: CbNeed evaluation—usually considered in the closed setting—is an implosive sharing refinement of Closed CbN. Its correctness is notoriously technical, see for instance Maraist, Odersky, and Wadler [46], and Ariola and Felleisen [16], [15]. Kesner develops an elegant alternative technique resting on multi types [43], that has been adapted to the strong case—becoming quite more technical—in [19]. The correctness of implementations of optimal reductions is extremely involved and sophisticated, see Asperti and Guerrini [17]. None of these works are presented using abstract machines. Call-by-need machines do exist, but their correctness is always proved relatively to a call-by-need calculus, as in [35], with respect to which they are not implosive—the standard correctness technique indeed applies.

Proofs: Proofs are in the Appendix. With respect to the LICS 2021 proceedings version of the paper, this version also contains a few more technical details in the body of the paper starting from Section VII.

II. The Value Substitution Calculus

Plotkin's call-by-value λ-calculus [50] is known to behave perfectly as long as terms are closed (that is, without free variables) and evaluation is weak—let us call such a setting *Closed CbV*, following Accattoli and Guerrieri [8].

It is well known that as soon as one considers open term or strong evaluation then Plotkin's CbV β_v-rule $(\lambda x.t)v \to_{\beta_v} t\{x\leftarrow v\}$ is no longer adequate with various semantical properties—as first shown by Paolini and Ronchi della Rocca [49], [48], [51]—and the operational semantics has to be extended somehow. In [8], Accattoli and Guerrieri compare various ways of doing it, and show that in the open setting they are all equivalent[1].

Here we adopt one of those calculi, Accattoli and Paolini's *value substitution calculus*, shortened here to VSC [11]. It was first introduced to study a semantical property of Strong CbV, *solvability*, and it is isomorphic to the CbV representation of the λ-calculus into linear logic, as shown by Accattoli [1].

It is also well known that values can be defined as variables and abstractions, or simply as abstractions—the resulting theories differ only for inessential details. Restricting values to abstractions is preferred by works on CbV abstract machines—including this one—because it leads to better performances, as shown by Accattoli and Sacerdoti Coen [13].

The Value Substitution Calculus: There are various ingredients in the VSC. First, the syntax of the λ-calculus is extended with let-expressions, that we here prefer to more compactly write as explicit substitutions $t[x\leftarrow u]$ (shortened to ES), while we use $t\{x\leftarrow u\}$ for meta-level substitution.

$$\begin{array}{rll} \text{VSC Values} & v & ::= \lambda x.t \\ \text{VSC Terms} & t,u,p & ::= x \mid \lambda x.t \mid tu \mid t[x\leftarrow u] \end{array}$$

[1] One of these calculi is (the CbV and intuitionistic fragment of) Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus [30], which could be used to reformulate the results in this work, another one is Guerrieri and Carraro's *shuffling calculus* λ_{shuf} [27], which instead could not, because its cost model is unclear, see [8].

There also is a crucial use of *contexts* to specify the rewriting rules. Contexts are terms with a *hole* $\langle\cdot\rangle$ intuitively standing for a removed subterm. We shall see various notion of contexts. For now, we need unrestricted contexts C and the special case of substitution contexts L (standing for *L*ist of substitutions).

$$\begin{array}{rl} \textsc{Contexts} & C ::= \langle\cdot\rangle \mid Ct \mid tC \mid \lambda x.C \mid C[x{\leftarrow}t] \mid t[x{\leftarrow}C] \\ \textsc{Sub. Ctxs} & L ::= \langle\cdot\rangle \mid L[x{\leftarrow}t] \end{array}$$

Replacing the hole of a context C with a term t (or another context C') is called *plugging* and noted $C\langle t\rangle$ (resp. $C\langle C'\rangle$).

Given the use of explicit substitutions (shortened to ES), β-steps are decomposed in two, the introduction of the ES and the turning of an ES into a meta-level substitution. The rewrite rules work up to a substitution context L, or, if you prefer, up to ES (also called *at a distance*).

$$\begin{array}{c} \textsc{VSC Rules at top level} \\ \begin{array}{rcl} \textsc{Multiplicative} & L\langle\lambda x.t\rangle u & \mapsto_{\mathsf{m}} L\langle t[x{\leftarrow}u]\rangle \\ \textsc{Exponential} & t[x{\leftarrow}L\langle v\rangle] & \mapsto_{\mathsf{e}} L\langle t\{x{\leftarrow}v\}\rangle \end{array} \\ \textsc{Contextual closure:} \quad \dfrac{t \mapsto_a t'}{C\langle t\rangle \to_a C\langle t'\rangle} \quad (a \in \{\mathsf{m},\mathsf{e}\}) \\ \textsc{Notation}: \quad \to_{\mathsf{vsc}} := \to_{\mathsf{m}} \cup \to_{\mathsf{e}} \end{array}$$

Examples: $(\lambda x.t)[y{\leftarrow}u]p \to_{\mathsf{m}} t[x{\leftarrow}p][y{\leftarrow}u]$ and $t[x{\leftarrow}v[y{\leftarrow}u]] \to_{\mathsf{e}} t\{x{\leftarrow}v\}[y{\leftarrow}u]$. The terminology comes from the connection with linear logic proof nets. Note that the CbV restriction is not on multiplicative/β-redexes, but on exponential redexes.

Please note that the VSC can simulate Plotkin's β_v rule, as $(\lambda x.t)v \to_{\mathsf{m}} t[x{\leftarrow}v] \to_{\mathsf{e}} t\{x{\leftarrow}v\}$. Actually, it does more: in the VSC an open term such as $t := (\lambda x.\delta)(yy)\delta$, where $\delta := \lambda x.xx$ is the duplicator, diverges as follows

$$\begin{array}{rll} t & \to_{\mathsf{m}} \delta[x{\leftarrow}yy]\delta & \to_{\mathsf{m}} \\ & (zz)[z{\leftarrow}\delta][x{\leftarrow}yy] & \to_{\mathsf{e}} (\delta\delta)[x{\leftarrow}yy] \to_{\mathsf{vsc}} \dots \end{array}$$

while for Plotkin it is normal.

A key property of the VSC is that while $\to_{\mathsf{vsc}}$ obviously does not terminate—being able to simulate Plotkin's β_v rule—its two rules when taken separately are strongly normalizing.

Lemma II.1 (Local termination, [11]). *The reductions $\to_{\mathsf{m}}$ and $\to_{\mathsf{e}}$ are strongly normalizing.*

An *evaluation* is a possibly empty sequence $d : t \to^*_{\mathsf{vsc}} u$ of $\to_{\mathsf{vsc}}$ steps, whose number of $\to_{\mathsf{m}}$ (resp. $\to_{\mathsf{e}}$) steps is noted $|d|_{\mathsf{m}}$ (resp. $|d|_{\mathsf{e}}$).

Next, we discuss the simple open fragment, as it allows to introduce some key concepts for the general case, and on top of which we shall define the parallel strategy to be implemented by the SCAM.

The Open VSC: The open fragment of the VSC is obtained by first defining open contexts—by removing the abstraction case—and then use them to define the open variant of the rewriting rules, which do not evaluate under abstraction.

$$\begin{array}{c} \textsc{Open ctxs} \quad O ::= \langle\cdot\rangle \mid Ot \mid tO \mid O[x{\leftarrow}t] \mid t[x{\leftarrow}O] \\ \textsc{Open rewrite rules:} \quad \dfrac{t \mapsto_a t'}{O\langle t\rangle \to_{\mathsf{o}a} O\langle t'\rangle} \quad (a \in \{\mathsf{m},\mathsf{e}\}) \\ \textsc{Open reduction}: \quad \to_{\mathsf{o}} := \to_{\mathsf{om}} \cup \to_{\mathsf{oe}} \end{array}$$

Careful: the open fragment contains closed terms, because terms and contexts are *potentially* (and not necessarily) open.

Note that the grammar of open contexts implies that evaluation is non-deterministic, as rewriting steps can take place on both sides of an application and on both subterms of ES. For instance, for any step $t \to_{\mathsf{o}} u$, we have the following span $ut \;{}_{\mathsf{o}}{\leftarrow}\; tt \to_{\mathsf{o}} tu$ that closes on uu with one $\to_{\mathsf{o}}$ step on each side—the same happens with $u[x{\leftarrow}t] \;{}_{\mathsf{o}}{\leftarrow}\; t[x{\leftarrow}t] \to_{\mathsf{o}} t[x{\leftarrow}u]$.

Such a non-determinism is harmless because it is *diamond*. A rewriting relation $\to$ is diamond if $u_1 \leftarrow t \to u_2$ and $u_1 \neq u_2$ imply $u_1 \to p \leftarrow u_2$ for some p (it is the 1-step strengthening of confluence).

Proposition II.2 ([11]). *The reduction $\to_{\mathsf{o}}$ is diamond.*

There are two famous consequences of being diamond: *uniform normalization*, that is, if there is a normalizing reduction sequence then there are no diverging sequences, and *random descent*, that is, when a term is normalizable, all sequences to normal form have the same length. Essentially, the diamond is a relaxed form of determinism.

The normal forms of the open fragment have a nice inductive characterization, coming from the so-called *fireball calculus* [8]. *Fireballs* are defined by mutual induction with *inert terms*, and including values, as follows.

$$\begin{array}{rrcl} \textsc{Inert terms} & i, i' & ::= & x \mid if \mid i[x{\leftarrow}i'] \\ \textsc{Fireballs} & f, f' & ::= & v \mid i \mid f[x{\leftarrow}i] \end{array}$$

For instance, $\lambda y.((\lambda x.y)y)$ is a fireball as a value, while x, $y(\lambda x.x)$, xy, and $(z(\lambda x.\Omega))(zz)$ are fireballs as inert terms.

Proposition II.3 ([11]). *Let t be a VSC term. t is $\to_{\mathsf{o}}$ normal if and only if t is a fireball.*

The Strong Calculus: Outside of the open fragment, evaluation is not necessarily diamond. For instance, for any step $t \to_{\mathsf{vsc}} u$, the following span

$$(\lambda y.t)(\lambda y.t) \;{}_{\mathsf{e}}{\leftarrow}\; (xx)[x{\leftarrow}\lambda y.t] \;\to_{\mathsf{vsc}}\; (xx)[x{\leftarrow}\lambda y.u]$$

closes on $(\lambda y.u)(\lambda y.u)$ but not with a diamond diagram. Anyway, the VSC is confluent.

Proposition II.4 ([11]). *The reduction $\to_{\mathsf{vsc}}$ is confluent.*

Strong normal forms also have a nice characterization, iterating inside values the one for open normal forms.

$$\begin{array}{rl} \textsc{Strong inert terms} & i_{\mathsf{s}} ::= x \mid i_{\mathsf{s}} f_{\mathsf{s}} \mid i_{\mathsf{s}}[x{\leftarrow}i'_{\mathsf{s}}] \\ \textsc{Strong values} & v_{\mathsf{s}} ::= \lambda x.f_{\mathsf{s}} \\ \textsc{Strong fireballs} & f_{\mathsf{s}} ::= i_{\mathsf{s}} \mid v_{\mathsf{s}} \mid f_{\mathsf{s}}[x{\leftarrow}i_{\mathsf{s}}] \end{array}$$

For instance, $\lambda y.(y(\lambda x.y))$ is a strong value, while $\lambda y.((\lambda x.y)y)$ is not. Similarly, $x(\lambda z.zz)$ is a strong inert term, while $x(\lambda z.((\lambda y.y)z))$ is not. Note that strong fireballs are similar to the normal forms of the (CbN) λ-calculus, except that they can have ES containing strong inert terms.

Lemma II.5 (Characterization of normal forms). *Let t be a VSC term. t is $\to_{\mathsf{vsc}}$-normal if and only if t is a strong fireball.*

Proof p. 16

III. The External Strategy

Since the VSC is not diamond, we need to isolate an evaluation strategy, playing the role of the leftmost(-outermost) strategy in CbN. Usually, the strategies implemented by abstract machines are deterministic. Here instead we adopt a *diamond* strategy, that—as we explained in the previous section—can be seen as a form of relaxed determinism.

While CbN has a clear left-to-right orientation, reflected by its leftmost evaluation strategy, in CbV there is no such direction of evaluation. For instance, Plotkin standard evaluations are left-to-right [50], Leroy's ZINC abstract machine [44] is right-to-left, and Dal Lago and Martini follow an unspecified non-deterministic order [31]. Our strategy shall then be liberal, and not impose an order on the evaluation of applications.

On the other hand, we shall keep the *outermost* aspect of the leftmost-outermost CbN strategy. Our *external* strategy, indeed, shall reduce only redexes that cannot be duplicated or erased by any other redex.

The definition of the strategy requires the auxiliary notion of rigid terms, which are the variation over inert terms where the arguments of the head variable can be whatever term.

$$\textsc{Rigid terms} \quad r, r' ::= x \mid rt \mid r[x\leftarrow r']$$

Every (strong) inert term is a rigid term, but the converse does not hold, consider for instance $y\,(\delta\delta)$.

External (evaluation) contexts are defined by mutual induction with *rigid contexts*.

$$\begin{array}{cccl} & \multicolumn{3}{c}{\textsc{Term evaluation contexts}} \\ \textsc{External} & E & ::= & \langle\cdot\rangle \mid \lambda x.E \mid t[x\leftarrow R] \mid E[x\leftarrow r] \mid R \\ \textsc{Rigid} & R & ::= & rE \mid Rt \mid R[x\leftarrow r] \mid r[x\leftarrow R] \end{array}$$

Finally, the rewriting rules are obtained by closing the open rules with external contexts.

$$\textsc{External rewrite rules:} \quad \frac{t \to_{\mathsf{o}a} t'}{E\langle t\rangle \to_{\mathsf{x}a} E\langle t'\rangle} \quad (a \in \{\mathsf{m}, \mathsf{e}\})$$

$$\textsc{External reduction:} \quad \to_{\mathsf{x}} := \to_{\mathsf{xm}} \cup \to_{\mathsf{xe}}$$

Key points:

- *Normalizing*: the strategy normalizes the potentially diverging term $(\lambda x.y)(\lambda z.\Omega) \to_{\mathsf{xm}} y[x\leftarrow\lambda z.\Omega] \to_{\mathsf{xe}} y$, and diverges on $y(\lambda z.\Omega)$. In a companion paper about the semantics of Strong CbV by Accattoli, Guerrieri, and Leberle [10], it is proved that the external strategy is normalizing, *i.e.*, it reaches a normal form whenever it exists in the VSC.
- *External*: external steps are not contained in any value that is applied or ready to be substituted. The grammars of external and rigid contexts indeed forbid these situations: given an open step $t \to_{\mathsf{o}} u$, note that $(\lambda x.t)p \not\to_{\mathsf{x}} (\lambda x.u)p$ and $(xx)[x\leftarrow\lambda y.t] \not\to_{\mathsf{x}} (xx)[x\leftarrow\lambda y.u]$. On the other hand, the external strategy does enter values that shall not be substituted, for instance $yp(\lambda x.t) \to_{\mathsf{x}} yp(\lambda x.u)$ and $p[x\leftarrow y(\lambda x.t)] \to_{\mathsf{x}} p[x\leftarrow y(\lambda x.u)]$.
- *Non-determinism*: since $\to_{\mathsf{x}}$ contains the open rules, it is neither left-to-right nor right-to-left—we have both $(\mathsf{II})(\mathsf{II}) \to_{\mathsf{xm}} (y[y\leftarrow\mathsf{I}])(\mathsf{II})$ and $(\mathsf{II})(\mathsf{II}) \to_{\mathsf{xm}} (\mathsf{II})(y[y\leftarrow\mathsf{I}])$. Another example is given by $t = x(\lambda y.(\mathsf{II}))[x\leftarrow w(\mathsf{II})] \to_{\mathsf{xm}} x(\lambda y.z[z\leftarrow\mathsf{I}])[x\leftarrow w(\mathsf{II})]$, and $t \to_{\mathsf{xm}} x(\lambda y.(\mathsf{II}))[x\leftarrow w(z[z\leftarrow\mathsf{I}])]$.

Proposition III.1 (Properties of $\to_{\mathsf{x}}$)**.** *Let t be a VSC term.* Proof p. 22

1) Diamond*: $\to_{\mathsf{x}}$ is diamond. Moreover, every $\to_{\mathsf{x}}$ evaluation to normal form (if any) has the same number of $\to_{\mathsf{xm}}$ steps.*
2) Normal forms*: if t is x-normal then it is a strong fireball.*

Cost Model of the VSC: As time cost model of the VSC we take the number of $\to_{\mathsf{m}}$ steps of the external strategy. At the end of the paper, we shall prove it reasonable. *Subtlety*: the cost model makes sense despite the non-determinism of $\to_{\mathsf{x}}$, because the diamond of $\to_{\mathsf{x}}$ in particular preserves the kind of step, and so all evaluations to normal form have the same number of $\to_{\mathsf{m}}$ steps, as stated above.

Structural Equivalence: The VSC comes with a notion of structural equivalence $\equiv$, that equates terms differing only for the position of ES. A strong justification comes from the CbV linear logic interpretation of λ-terms with ES, in which structurally equivalent terms translate to the same (recursively typed) proof net, see [1].

The SCAM shall implement the external strategy $\to_{\mathsf{x}}$, but only up to $\equiv$, which is why we introduce $\equiv$ here.

Structural equivalence $\equiv$ is defined as the least equivalence relation on terms closed by all contexts and generated by the following top-level cases:

$$\begin{array}{rll} t[y\leftarrow p][x\leftarrow u] \equiv_{\mathsf{com}} t[x\leftarrow u][y\leftarrow p] & \text{if } y \notin \mathsf{fv}(u),\ x \notin \mathsf{fv}(p) \\ t\,p[x\leftarrow u] \equiv_{@\mathsf{r}} (tp)[x\leftarrow u] & \text{if } x \notin \mathsf{fv}(t) \\ t[x\leftarrow u[y\leftarrow p]] \equiv_{[\cdot]} t[x\leftarrow u][y\leftarrow p] & \text{if } y \notin \mathsf{fv}(t) \\ t[x\leftarrow u]p \equiv_{@\mathsf{l}} (tp)[x\leftarrow u] & \text{if } x \notin \mathsf{fv}(p) \end{array}$$

Extending the VSC with $\equiv$ results in a smooth system, as $\equiv$ commutes with evaluation, and can thus be postponed. Additionally, the commutation is *strong*, as it preserves the number and kind of steps (thus the cost model)—one says that it is a *strong bisimulation* (with respect to $\to_{\mathsf{x}}$). In particular, the equivalence is not needed to compute and it does not break, or make more complex, any property of the calculus—on the contrary, it makes it more flexible.

Proposition III.2 ($\equiv$ is a strong bisimulation)**.** *If $t \equiv u$ and $t \to_{\mathsf{a}} t'$ then there exists $u' \in \Lambda_{\mathsf{vsc}}$ such that $u \to_{\mathsf{a}} u'$ and $t' \equiv u'$, for $\mathsf{a} \in \{\mathsf{m}, \mathsf{e}, \mathsf{om}, \mathsf{oe}, \mathsf{xm}, \mathsf{xe}\}$.* Proof p. 23

Note that Prop. III.2 implies that $\equiv$ preserves normal forms.

IV. A Taste of Useful and Implosive Sharing

Here we use the VSC to give an informal overview of the various forms of sharing at work in this work.

The VSC comes with ES $t[x\leftarrow u]$, which are a form of *subterm sharing*. The exponential rewriting rule, however, rests on meta-level substitution, and so the system is closer to the λ-calculus than to an implementation, which would rather use a micro-step variant of the exponential rule such as

$$\textsc{Micro Exponential}$$
$$C\langle x\rangle[x\leftarrow L\langle v\rangle] \quad \to_{\mathsf{mi\text{-}e}} \quad L\langle C\langle v\rangle[x\leftarrow v]\rangle$$

Useful Sharing: In CbN, useful sharing amounts to two modifications of the substitution process, which are mandatory for reasonable implementations of strong evaluation. They are motivated by two paradigmatic cases of size explosions, one related to open terms and one to strong evaluation.

The first example of size-explosion is given by the family of open terms $\{t_n y\}_{n=1,2,\dots}$ and the family $\{u_n\}_{n=1,2,\dots}$ of their normal forms (in the ordinary λ-calculus) which are inert terms, where t_n and u_n are defined as follows:

$$\begin{array}{rcl|rcl} t_1 & := & \delta = \lambda x.xx & u_1 & := & yy \\ t_{n+1} & := & \lambda x.t_n(xx) & u_{n+1} & := & u_n u_n \end{array}$$

We use $|t|$ for the size of a term. Size explosion is proved via an auxiliary property. It is worth noticing that in this example the explosion is independent of the evaluation strategy.

Proposition IV.1 (Open and strategy-independent size explosion)**.** *Let $n, m > 0$.*

1) Auxiliary property*: $t_n u_m \to_\beta^n u_{n+m}$.*
2) $t_n y \to_\beta^n u_n$, $|t_n| = \mathcal{O}(n)$, $|u_n| = \Omega(2^n)$.

This case of explosion is avoided by useful sharing by forbidding substitutions of normal terms that are not abstractions, because they do not create β-redexes—note that the evaluation of the family substitutes y or instances of u_i, which are inert terms and thus not abstractions. In Strong CbV as presented via the VSC, this is hardcoded, as only abstractions can be substituted, so nothing needs to be changed. The effect of the optimization can be seen on normal forms: it is accounted by the fact that strong fireballs have ES containing strong inert terms, which are exactly normal terms that are not abstractions.

The second example of size-explosion is a closed variant of the first one, due to Accattoli [2]. Define:

$$\begin{array}{rcl|rcl} q_1 & := & \lambda x.\lambda y.yxx & p_0 & := & \mathsf{I} = \lambda z.z \\ q_{n+1} & := & \lambda x.q_n(\lambda y.yxx) & p_{n+1} & := & \lambda y.yp_np_n \end{array}$$

Proposition IV.2 (Closed and strategy-independent size explosion, [2])**.** *Let $n > 0$. Then $q_n\mathsf{I} \to_\beta^n p_n$. Moreover, $|q_n\mathsf{I}| = \mathcal{O}(n)$, $|p_n| = \Omega(2^n)$, $t_n\mathsf{I}$ is closed, and p_n is normal.*

In this second case, substituting only abstractions does not help, because the terms that are substituted along the evaluation are the identity I and instances of p_i, which are all abstractions. If evaluation is weak, and substitution is done micro-step, then there is no problem because the replaced variables are all instances of x in some q_i, which are under abstraction and which are never replaced in micro-step weak evaluation. With micro-step strong evaluation, however, these replacement do happen, and the size explodes.

To tame this problem, useful sharing rests on an optimization sometimes called *substituting abstractions on-demand*, which is trickier. It requires abstractions to be substituted *only* on applied variable occurrences: note that the explosion is caused by replacements of variables (namely the instances of x) which are *not* applied, and that thus do not create β-redexes. A step such as $(xy)[x\leftarrow v] \to_{\mathsf{e}} vy$ is accepted, or, *it is useful*, because it creates a β/multiplicative redex, while a step such as $(yx)[x\leftarrow v] \to_{\mathsf{e}} yv$ is *useless*, and must not be done.

Note however that this optimization makes sense only when one switches to micro-step evaluation via $\to_{\mathsf{mi\text{-}e}}$ above, that is, at the level of machines, because in $(xx)[x\leftarrow v]$ there are both a useful and a useless occurrence of x. The implementation of *substituting abstractions on-demand* is very subtle, also because by not performing useless substitutions, it leaves pending ES with values.

Mixing Implosive and Useful Sharing: There is a case concerning such pending ES where there is some freedom in deciding how to evaluate. Consider a term such as $((xy)y)[y\leftarrow\lambda z.t]$ where t is a term with some β-redexes. Both micro-step substitutions of $\lambda z.t$ on y are useless, but t needs to be evaluated. The non-implosive choice is to copy $\lambda z.t$, obtaining $(x(\lambda z.t))(\lambda z.t)$, and then evaluate t twice. This is what Accattoli and Dal Lago do in their useful implementations of the leftmost-outermost CbN strategy in [7], [3]. It can be seen as useful, because each copy of $\lambda z.t$ contains some β-redexes, so one is not substituting for nothing.

The implosive choice, which is also more CbV in spirit, is to evaluate $\lambda z.t$ only once, keeping it in the ES, that is, reducing $((xy)y)[y\leftarrow\lambda z.t]$ to some $((xy)y)[y\leftarrow\lambda z.u]$. This is what the SCAM shall do.

A natural question: Why not *always* evaluate values before substituting them? Because, it is unsound with respect to normalization in the Strong CbV. Consider $t := (x(\lambda x.y))[x\leftarrow\lambda z.z(\lambda w.\Omega)]$. The term t would then diverge because it would evaluate Ω, while it normalizes to y, for instance with the external strategy:

$$\begin{array}{l}(x(\lambda x.y))[x\leftarrow\lambda z.z(\lambda w.\Omega)] \to_{\mathsf{xe}} (\lambda z.z(\lambda w.\Omega))(\lambda x.y) \\ \to_{\mathsf{xm}}\to_{\mathsf{xe}} (\lambda x.y)(\lambda w.\Omega) \to_{\mathsf{xm}}\to_{\mathsf{xe}} y\end{array}$$

Note that in this example the substitution happens on an applied variable. Evaluating a value v before substituting it can be done safely only when in $t[x\leftarrow v]$ the term t is normal and all the occurrences of x in t are not applied (that is, the associated micro substitution steps are useless), which is exactly when the SCAM shall do it.

V. Relaxed Implementations

Here we explain abstractly the subtle and unusual way in which our machine implements the external strategy $\to_{\mathsf{x}}$ modulo structural equivalence $\equiv$.

Before giving the details, let us stress a key point. The machine is started on λ-terms, not VSC terms, that is, the initial term is not supposed to have any ES. Non-initial states of the machine however shall decode to VSC terms.

Machines and Structural Strategies: A *machine* $\mathsf{M} = (s, \rightsquigarrow, \cdot^\circ, \cdot\downarrow)$ is a transitions system $\rightsquigarrow$ over a set of states, noted s, with transitions partitioned into *β-transitions* $\rightsquigarrow_\beta$ and *overhead transitions* $\rightsquigarrow_{\mathsf{o}}$, together with a *compilation* function $\cdot^\circ$ turning λ-terms into states, and a read-back function $\cdot\downarrow$ turning states into VSC terms and satisfying the *initialization constraint* $t^\circ\downarrow = t$ for all λ-terms t. A state s is *initial* if $s = t^\circ$ for some λ-term t, and *final* if no transitions apply. An *execution* $\rho : s \rightsquigarrow^* s'$ is a possibly empty sequence of transitions from an initial state to a state s' said *reachable*.

A *structural strategy* $(\to, \equiv)$ is a rewriting relation $\to$ together with a structural equivalence $\equiv$ on VSC terms, such that $\equiv$ is a *strong bisimulation* with respect to $\to$.

Relaxed Implementations: In the literature, a machine implements a (structural) strategy when the two are weakly bisimilar, where weakness is given by the fact that the overhead transitions of the machine (that search for redexes and decompose the substitution process) are invisible on the calculus. The bisimulation relates executions of the machine and evaluations on the calculus *locally*, or *small-step*, that is, β-step-by-β-step, and for sequences not necessarily reaching a normal form. In particular, there is a bijection between the β-steps of the strategy and the β-transitions of the machine.

Our machine does not follow such a simple schema, because it evaluates the body of some shared abstractions, and each β-transition in these bodies potentially maps (via read-back) to *many* β-steps on the calculus, breaking the bijection, and forbidding the machine to simulate single steps of the calculus.

We then adopt a relationship between the strategy and the machine that is weaker than a bisimulation and asymmetric: the strategy simulates the machine locally (potentially taking many steps for each β-transition), while the machine simulates the strategy only *globally*, or *big-step*: preserving divergence and normalizing evaluations, with their cost model, but not β-step-by-β-step. In the VSC, the role of β steps on the calculus is played by multiplicative steps, whose number in an evaluation sequence d is noted $|d|_{\mathsf{m}}$.

Definition V.1 (Relaxed implementations)**.** *A machine* $\mathsf{M} = (s, \rightsquigarrow, \cdot^\circ, \cdot\downarrow)$ *is a* relaxed implementation of a structural strategy $(\to, \equiv)$ *on VSC terms when, given a* λ*-term* t*:*

1) Executions to evaluations*: for any* M*-execution* $\rho : t^\circ \rightsquigarrow^*_{\mathsf{M}} s$ *there is a* $\to$*-evaluation* $d : t \to^* \equiv s\downarrow$ *with* $|\rho|_\beta \leq |d|_{\mathsf{m}}$.
2) Normalizing evaluations to executions*: if* $d\colon t \to^* u$ *with* u $\to$*-normal then there is an* M*-execution* $\rho\colon t^\circ \rightsquigarrow^*_{\mathsf{M}} s$ *with* s *final such that* $s\downarrow \equiv u$ *with* $|\rho|_\beta \leq |d|_{\mathsf{m}}$.
3) Diverging evaluations to executions*: if* $\to$ *diverges on* t *then* M *diverges on* t° *doing infinitely many* β*-transitions.*

Next, we isolate sufficient conditions for relaxed implementations, that shall structure our implementative study.

Definition V.2 (Relaxed implementation system)**.** *A relaxed implementation system is given by a* machine $\mathsf{M} = (s, \rightsquigarrow, \cdot^\circ, \cdot\downarrow)$ *and a* structural strategy $(\to, \equiv)$ *such that for every reachable state* s*:*

1) Relaxed β-projection*:* $s \rightsquigarrow_\beta s'$ *implies that there exists* $d : s\downarrow \to^+ \equiv s'\downarrow$ *such that* $|d|_{\mathsf{m}} \geq 1$*;*
2) Overhead transparency*:* $s \rightsquigarrow_{\mathsf{o}} s'$ *implies* $s\downarrow \equiv s'\downarrow$*;*
3) Overhead transitions terminate*:* $\rightsquigarrow_{\mathsf{o}}$ *terminates;*
4) Halt*: if* s *is final then* $s\downarrow$ *is* $\to$*-normal;*
5) Lax determinism*:* $\to$ *is diamond and* $\rightsquigarrow$ *is deterministic.*

Theorem V.3 (Abstract implementation)**.** *Let* M *and* $(\to, \equiv)$ *form a relaxed implementation system. Then,* M *is a relaxed implementation of* $(\to, \equiv)$.

Proof p. 23

VI. Introducing the SCAM

In the next section we start with the implementative details: let us overview some key points first.

Crumbling: The SCAM builds on the theory of CbV abstract machines developed by Accattoli and co-authors [4], [12], [9], [6]. In particular, it relies on the *crumbling* technique of [6], which essentially is a specific presentation of the transformation into *administrative normal forms* by Flanagan et al. [38], [52]. As shown in [6], crumbling allows to reduce the number of data structures required, as it encodes the dump and the stack of CbV machines inside the environment. This in turn reduces the number of transitions of the machine. Both aspects are extremely valuable when studying strong evaluation, as strong machines tend to have many data structures and at least a dozen transitions. Our *strong crumbling abstract machine* (shortened to SCAM)—thanks to crumbling—is compact, having only 1 data structure and 9 transitions. The price to pay are the technicalities of crumbling, roughly amounting to a light form of compilation.

Garbage Collection: An unusual but key aspect is that garbage collection is done by the SCAM itself, that is, it is not left to the meta-level garbage collector, as it is usually the case with abstract machines. This happens because, in the search for values to evaluate strongly, the SCAM has to avoid the garbage ones, because evaluating their bodies would indeed break correctness with respect to Strong CbV.

Zig-Zag: The SCAM shall have two alternating phases, one performing open evaluation, and one searching for a value $\lambda x.t$ to evaluate strongly. Once the SCAM finds it, it switches to the open phase for evaluating its body t, and so on. The open phase can be implemented exploring the code from left-to-right or from right-to-left—we adopt right-to-left, because this choice induces some stronger invariants. The phase searching for values is instead left-to-right, because it also performs garbage collection, which cannot be done right-to-left. Our mixed order is hinted at in Biernacka et al. [22] as a possible optimization of their right-to-left strong machine.

Two Levels of Implosive Sharing: There are two levels of implosiveness, connected to the out/under abstraction dichotomy. *Shallow implosive sharing* evaluates inside shared subterms but not inside shared abstractions. This happens in CbNeed. *Deep implosive sharing*, instead, also enters shared abstractions—an instance is optimal reduction[2]. The SCAM adopts a deep implosive approach to useful sharing.

VII. Compilation and Read-Back

In a CbV λ-calculus with a construct for subterm sharing, such as ES, applications can be decomposed by introducing sharing points for any non-variable subterm. Here we consider the case where applications are only between

[2]We avoid the weak/strong terminology, because there can be strong evaluation with shallow implosive sharing, as in Strong Call-by-Need [19].

AUXILIARY			CRUMBLING		
$\overline{x} \coloneqq (x, \epsilon)$	$\overline{\lambda x.t} \coloneqq (z, [z{\leftarrow}\lambda x.\underline{t}])$	$\overline{tu} \coloneqq (z, [z{\leftarrow}xy]ee')$	$\underline{x} \coloneqq [\star{\leftarrow}x]$	$\underline{\lambda x.t} \coloneqq [\star{\leftarrow}\lambda x.\underline{t}]$	$\underline{tu} \coloneqq [\star{\leftarrow}xy]ee'$

where $\overline{t} = (x, e)$ and $\overline{u} = (y, e')$ in both $\overline{tu}$ and $\underline{tu}$; and z fresh in V_{cr} in both $\overline{\lambda x.t}$ and $\overline{tu}$.

Fig. 1: Crumbling transformation.

variables[3]. For instance, the crumbling representation $\underline{t}$ of $t \coloneqq (\lambda x.(\lambda y.y)\,(xx))\,(\lambda z.zz)$ (see forthcoming Ex. VII.2) is

$$(\mathtt{x}\mathtt{y})[\mathtt{x}{\leftarrow}\lambda x.\mathtt{z}\mathtt{w}[\mathtt{z}{\leftarrow}\lambda y.y][\mathtt{w}{\leftarrow}xx]][\mathtt{y}{\leftarrow}\lambda z.zz]$$

where we denoted the sharing points introduced by the transformation by $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{w}$. Note that the transformation involves also function bodies (*i.e.* $\lambda x.(\lambda y.y)\,(xx)$ turns into $\lambda x.(\mathtt{z}\mathtt{w})[\mathtt{z}{\leftarrow}\lambda y.y][\mathtt{w}{\leftarrow}xx]$), that ES are grouped together unless forbidden by abstractions, and that ES are flattened out, i.e. they are not nested unless nesting is forced by abstractions. Here we shall adopt a variant of this transformation, having the first subterm $\mathtt{x}\mathtt{y}$ of $\underline{t}$ in a pending ES $[\star{\leftarrow}\mathtt{x}\mathtt{y}]$ on a special variable $\star$ dedicated to such pending ES—this is analogous to the initial continuation of continuation-passing transformations.

Such a *crumbled representation* of terms impacts on the design of machines for CbV evaluation. By removing the applicative structure, there is no need for data structures encoding the evaluation context, such as the applicative stack and the dump, that get encoded in the environment. The environment is the data structure for sharing that collects the ES obtained 1) at compile time, *i.e.* by the crumbling transformation and 2) dynamically, during execution.

In [6], it is shown that the crumbling technique smoothly accommodates open terms, by designing an abstract machine that implements Open CbV within a bilinear overhead (when implemented on RAM). This paper extends that work to the strong case, but as explained in the introduction, the extension is non-trivial. We now cover compilation via crumbling; the next section deals with the open machine, and Sect. IX presents the strong extension.

Crumbled Environments: We first have to define the target language of the translation, which are not terms with ES but *crumbled environments*, a slight variant. A crumbled environment is a list of ES containing *bites*, defined below. A key point is that we need to distinguish the variables introduced by the crumbling transformation from those originally in the term, which is why variables range over a set of names $V = V_{\mathsf{cr}} \uplus V_{\mathsf{calc}}$ where V_{cr} is the set of crumbling variables and V_{calc} the set of variables of the calculus, both infinite—the names in V_{cr} are sometimes noted $\mathtt{x}, \mathtt{y}, \mathtt{z}$ for clarity, but in general names from both sets are noted x, y, z. Moreover, there is a distinguished variable $\star \in V_{\mathsf{cr}}$.

$$\begin{array}{rrcll} \textsc{Bites} & b & ::= & x \mid xy \mid \lambda x.e & (*) \\ \textsc{(Crumbled) Envs} & e & ::= & \epsilon \mid e : [x{\leftarrow}b] & \end{array}$$

[3]For crumbling, we follow [6]. Therein applications can have abstractions as subterms. Here however we adopt the minor variant where abstractions are also removed from applications and shared.

Side conditions $(*)$: $x \neq \star \neq y$ in x and xy, and $x \in V_{\mathsf{calc}}$ and e is non-empty in $\lambda x.e$. The conditions imply that $\star$ cannot have free occurrences. As for terms, bites of the form $\lambda x.e$ are *values*, ranged over by v, while x and xy are *inert bites*.

Environments are defined concatenating on the right, but we shall freely concatenate also on the left, concatenate whole environments, and omit the concatenation symbol ':'. Environments are also meant to be looked up for substitution. *Notation:* $e(x) = b$ if $e = e'[x{\leftarrow}b]e''$ with $x \notin \mathsf{dom}(e')$, and $e(x) = \bot$ otherwise—note that in open/strong settings environments may be undefined on some variables.

Crumbling λ-Terms: Machines start their execution on the compilation of ordinary λ-terms (with no ES), and the following crumbling transformation $\underline{t}$ shall be our notion of compilation. Note that $\star$ appears always and only as the variable "bound" by the leftmost ES in $\underline{t}$.

Definition VII.1 (Crumbling transformation $\underline{\cdot}$)**.** *Let t be a λ-term. We define its crumbling $\underline{t}$ using an auxiliary function $\overline{\cdot}$ mapping λ-terms to pairs of a variable plus an environment. The formal definition of $\underline{\cdot}$ and $\overline{\cdot}$ are given in Fig. 1, and explained in the next example.*

Example VII.2. *The main transformation $\underline{\cdot}$ is used at top level, both of the initial term and recursively at top level of every function body. The auxiliary transformation $\overline{\cdot}$ instead is used when compiling applications, and it returns the variable that shall be used in place of the original term, plus a crumbled environment that binds additional results of the transformation.*

For the sake of example, let us consider the term

$$t \coloneqq (\lambda x.I\,(x\,x))\,\delta = (\lambda x.(\lambda y.y)\,(x\,x))\,(\lambda z.z\,z).$$

The term t consists of an application of two non-variable terms, hence the transformation yields

$$\underline{t} \quad = \quad [\star{\leftarrow}\mathtt{x}\mathtt{y}][\mathtt{x}{\leftarrow}\textbf{\textit{?}}]\cdots[\mathtt{y}{\leftarrow}\textbf{\textit{?}}]\cdots$$

where $\mathtt{x}$ and $\mathtt{y}$ are two fresh variables generated respectively by $\overline{\lambda x.I\,(xx)}$ and $\overline{\delta}$:

$$\begin{array}{rl} \overline{\lambda x.I\,(xx)} = & (\mathtt{x}, [\mathtt{x}{\leftarrow}\lambda x.\underline{I\,(xx)}]) \\ = & (\mathtt{x}, [\mathtt{x}{\leftarrow}\lambda x.\overline{[\star{\leftarrow}\mathtt{z}\mathtt{w}]}[\mathtt{z}{\leftarrow}\lambda y.[\star{\leftarrow}y]][\mathtt{w}{\leftarrow}xx]]) \\ \text{of the form} & (\mathtt{x}, [\mathtt{x}{\leftarrow}\lambda x.[\star{\leftarrow}\mathtt{z}\mathtt{w}][\mathtt{z}{\leftarrow}\textbf{\textit{?}}]\cdots[\mathtt{w}{\leftarrow}\textbf{\textit{?}}]\cdots]) \end{array}$$

$$\overline{\delta} \quad = (\mathtt{y}, [\mathtt{y}{\leftarrow}\lambda z.\underline{zz}]) = (\mathtt{y}, [\mathtt{y}{\leftarrow}\lambda z.[\star{\leftarrow}zz]]).$$

The fully transformed $\underline{t}$ is:

$$[\star{\leftarrow}\mathtt{x}\mathtt{y}][\mathtt{x}{\leftarrow}\lambda x.[\star{\leftarrow}\mathtt{z}\mathtt{w}][\mathtt{z}{\leftarrow}\lambda y.[\star{\leftarrow}y]][\mathtt{w}{\leftarrow}xx]][\mathtt{y}{\leftarrow}\lambda z.[\star{\leftarrow}zz]].$$

Names: A key point is that, as it is standard for abstract machines, crumbled environments and bites are *not* considered modulo α-equivalence. Some machine transitions shall rename variables: α-equivalence can rename x with y only if they are both in $V_{\mathsf{cr}} \setminus \{\star\}$ (resp. both in V_{calc}), and $\star$ cannot be renamed. We also need a notion of well-namedness for both λ-terms and environments.

Definition VII.3 (Well-named)**.** *A λ-term t is* well-named *if its bound variables are all distinct, and $\mathsf{fv}(t) \cap \mathsf{bv}(t) = \emptyset$. An environment e (resp. a bite b) is* well-named *if when two binders bind the same variable x then $x = \star$, and $\mathsf{fv}(e) \cap \mathsf{bv}(e) \subseteq \{\star\}$ (resp. $\mathsf{fv}(b) \cap \mathsf{bv}(b) \subseteq \{\star\}$).*

Read-back: Bites and environments are mapped to VSC terms via a read-back function, that in particular inverts the crumbling transformation. The distinction between the two kinds of variables plays a role. The intuition is that ES are unfolded when they come from crumbling *or* when they contain values, as to include in the read-back the useless part of the work done by $\to_{\mathsf{e}}$ on the calculus.

The read-back of a bite b and an environment e are, respectively, the terms $b{\downarrow}$ and $e{\downarrow}$ defined by:

BITES READ-BACK

$$x{\downarrow} := x \qquad (xy){\downarrow} := xy \qquad (\lambda x.e){\downarrow} := \lambda x.e{\downarrow}$$

CRUMBLED ENVIRONMENTS READ-BACK

$$\epsilon{\downarrow} := \star \quad e[x{\leftarrow}b]{\downarrow} := \begin{cases} e{\downarrow}\{x{\leftarrow}b{\downarrow}\} & \text{if } b = v \text{ or } x \in V_{\mathsf{cr}} \\ e{\downarrow}[x{\leftarrow}b] & \text{otherwise.} \end{cases}$$

Proof p. 29 **Lemma VII.4** (Crumbling properties)**.** *If t is a well-named λ-term then $\underline{t}$ is well-named and $\underline{t}{\downarrow} = t$.*

In the next sections, we shall need a modular deconstruction of read-back, spelled out below.

Definition VII.5. *Let e be an environment. Then the* substitution σ_e *and the* substitution context L_e induced by e *are given by (where $(*)$ stands for "$b = v$ or $x \in V_{\mathsf{cr}}$")*

SUBSTITUTION σ_e INDUCED BY e

$$\sigma_\epsilon := Id \qquad \sigma_{e[x{\leftarrow}b]} := \begin{cases} \sigma_e\{x{\leftarrow}b{\downarrow}\} & \text{if } (*) \\ \sigma_e & \text{otherw.} \end{cases}$$

SUBSTITUTION CONTEXT L_e INDUCED BY e

$$L_\epsilon := \langle\cdot\rangle \qquad L_{e[x{\leftarrow}b]} := \begin{cases} L_e\{x{\leftarrow}b{\downarrow}\} & \text{if } (*) \\ L_e[x{\leftarrow}b] & \text{otherw.} \end{cases}$$

Proof p. 31 **Lemma VII.6** (Modular read-back)**.** $(ee'){\downarrow} = L_{e'}\langle e{\downarrow}\sigma_{e'}\rangle$.

Example VII.7. *Let us consider the environment*

$$\underbrace{[\star{\leftarrow}y]}_{e}\ \underbrace{[y{\leftarrow}\mathtt{yy}][\mathtt{y}{\leftarrow}\lambda z.[\star{\leftarrow}z\ z]]}_{e'}$$

where y is a "normal" variable, and $\mathtt{y}$ is a crumbling variable. The read-back $ee'{\downarrow}$ proceeds as follows:

$$\begin{aligned} ee'{\downarrow} &= [\star{\leftarrow}y][y{\leftarrow}\mathtt{yy}]{\downarrow}\ \{\mathtt{y}{\leftarrow}\lambda z.[\star{\leftarrow}z\ z]\} \\ &= [\star{\leftarrow}y]{\downarrow}\ [y{\leftarrow}\mathtt{yy}]\{\mathtt{y}{\leftarrow}\lambda z.[\star{\leftarrow}z\ z]\} \\ &= \big([\star{\leftarrow}y]{\downarrow}\{\mathtt{y}{\leftarrow}\lambda z.[\star{\leftarrow}z\ z]\}\big)\ ([y{\leftarrow}\mathtt{yy}]\{\mathtt{y}{\leftarrow}\lambda z.[\star{\leftarrow}z\ z]\}) \end{aligned}$$

The equality in Lemma VII.6 holds at this point, since:

$$L_{e'} = \langle\cdot\rangle[y{\leftarrow}\mathtt{yy}]\{\mathtt{y}{\leftarrow}\lambda z.[\star{\leftarrow}z\ z]\} \qquad \sigma_{e'} = \{\mathtt{y}{\leftarrow}\lambda z.[\star{\leftarrow}z\ z]\}$$

The full read-back $ee'{\downarrow}$ is $y[y{\leftarrow}(\lambda z.[\star{\leftarrow}z\ z])(\lambda z.[\star{\leftarrow}z\ z])]$.

VIII. The Open Crumbling Machine

Here we overview an abstract machine implementing the open VSC $\to_{\mathsf{o}}$, that shall be the starting point for the strong machine of the next section. We keep following the crumbling technique by [6], slightly adapted.

The only structure at work in the machine is a crumbled environment, traversed from right to left, together with a pointer to where the machine is operating. Then a machine state $s := e \triangleleft e'$ is a pair of crumbled environments where

- *Right*: e' is the part that has already been processed,
- *Left*: e is the part yet to be processed, and
- *Separator*: $\triangleleft$ represents the pointer to the active point.

The Open Crumbling Abstract Machine (OCAM) has 4 transitions, two β transitions $\rightsquigarrow_{\beta_v}$ and $\rightsquigarrow_{\beta_i}$, and two overhead transitions $\rightsquigarrow_{\mathsf{ren}}$ and $\rightsquigarrow_{\mathsf{sea}}$, detailed below. Compilation is defined as $t^\circ := \underline{t} \triangleleft \epsilon$ for a well-named λ-term t, and read-back simply as $(e \triangleleft e'){\downarrow} := (ee'){\downarrow}$. By Lemma VII.4, compilation and read-back verify the initialization constraint for the OCAM.

β-Transitions: They are quite technical unfortunately, because of crumbling. The idea is that there are two cases, $\rightsquigarrow_{\beta_v}$ for when the argument is a value and $\rightsquigarrow_{\beta_i}$ for when it is a inert term. In the first case, the machine also does in one single transition both the β/multiplicative step and the exponential step that is created. Because of crumbling, the β-redex is given by an application of variables yz, whose abstraction and argument are to be found in the environment. Actually, the transitions also does the copy of the abstraction that replaces y. They are (further explanations follow):

$$\begin{array}{llll} e[x{\leftarrow}y\ z] \triangleleft e' & \rightsquigarrow_{\beta_v} & e([x{\leftarrow}b]e''\{w{\leftarrow}z\}) \triangleleft e' \\ e[x{\leftarrow}y\ z] \triangleleft e' & \rightsquigarrow_{\beta_i} & e[x{\leftarrow}b]e'' \triangleleft [w{\leftarrow}z]e' \end{array}$$

where in both cases $e'(y)$ is a value and $(e'(y))^\alpha =: \lambda w.([\star{\leftarrow}b]e'')$ is a well-named copy of $e'(y)$ with fresh names, and $e'(z)$ is a value in $\rightsquigarrow_{\beta_v}$, while in $\rightsquigarrow_{\beta_i}$ it is an inert bite.

Explanation. First, we explain points that are common to both transitions. The bite under analysis is yz and y maps to a value v in e', thus the read-back turns yz into $(yz)\sigma_{e'} = \sigma_{e'}(y)\sigma_{e'}(z) = v\sigma_{e'}(z)$ that is a β/multiplicative redex. The machine copies v, obtaining $\lambda w.([\star{\leftarrow}b]e'')$, as copying corresponds to α-renaming. Variables have indeed to be intended as memory locations, and α-renaming means making a copy somewhere else in the memory. Letting the argument z aside, what happens to both transitions is: the β-redex is fired and yz is replaced by the body $[\star{\leftarrow}b]e''$ of the copied value, that is concatenated with e, obtaining $e[x{\leftarrow}b]e''$. Via read-back, the multiplicative redex is $v\sigma_{e'}(z) = (\lambda w.t)\sigma_{e'}(z)$ with $t = ([\star{\leftarrow}b]e''){\downarrow}$, which takes a $\mapsto_{\mathsf{m}}$ step to $t[w{\leftarrow}\sigma_{e'}(z){\downarrow}]$.

Consider now the argument z. If it is associated with a value v in e' then its read-back is also a value, namely $v' = v\sigma_{e'}$, and so on the calculus $t[w{\leftarrow}v']$ is a $\mapsto_{\mathsf{e}}$ redex. The machine

$$
\begin{array}{rcrl}
e[x\leftarrow y\,z] \triangleleft K & \rightsquigarrow_{\beta_v} & e([x\leftarrow b]e'\{w\leftarrow z\}) \triangleleft K & \text{if } (*) \text{ and } e_K(z) = v \text{ for some } v;\\
e[x\leftarrow y\,z] \triangleleft K & \rightsquigarrow_{\beta_i} & e[x\leftarrow b]e' \triangleleft K\langle\langle\cdot\rangle[w\leftarrow z]\rangle & \text{if } (*) \text{ and } e_K(z) = i \text{ for some } i;\\
e[x\leftarrow y] \triangleleft K & \rightsquigarrow_{\mathsf{ren}} & e\{x\leftarrow y\} \triangleleft K & \text{if } x \neq \star;\\
e[x\leftarrow b] \triangleleft K & \rightsquigarrow_{\mathsf{sea}_1} & e \triangleleft K\langle\langle\cdot\rangle[x\leftarrow b]\rangle & \text{if none of the other rules is applicable.}
\end{array}
$$

$$
\begin{array}{rcrl}
\epsilon \triangleleft K & \rightsquigarrow_{\mathsf{sea}_2} & \epsilon \triangleright K & \\
e \triangleright K\langle\langle\cdot\rangle[x\leftarrow b]\rangle & \rightsquigarrow_{\mathsf{sea}_3} & e[x\leftarrow b] \triangleright K & \text{if } b \text{ is not a value;}\\
e \triangleright K\langle\langle\cdot\rangle[x\leftarrow v]\rangle & \rightsquigarrow_{\mathsf{gc}} & e \triangleright K & \text{if } x \notin \mathsf{fv}(e) \text{ and } e \text{ is not empty;}\\
e \triangleright K\langle e'[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle & \rightsquigarrow_{\mathsf{sea}_4} & e'[x\leftarrow\lambda y.e] \triangleright K & \\
e \triangleright K\langle\langle\cdot\rangle[x\leftarrow\lambda y.e']\rangle & \rightsquigarrow_{\mathsf{sea}_5} & e' \triangleleft K\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle & \text{if } x \in \mathsf{fv}(e).
\end{array}
$$

Fig. 2: Open (above) and strong (below) phases of the SCAM.

substitutes z for w in the body $[x\leftarrow b]e''$ of the copied value. This corresponds to performing the $\mapsto_{\mathsf{e}}$-redex $t[w\leftarrow v'] \mapsto_{\mathsf{e}} t\{w\leftarrow v'\}$ on the calculus. Note that the machine only performs a renaming, it does not duplicate v'—up to read-back this is equivalent. If instead z is associated to an inert bite in e', let us assume for a moment that z reads back to an inert term. Then $t[w\leftarrow\sigma_{e'}(z)\downarrow] = t[w\leftarrow i]$ for some inert term i, and no substitution happens.

For $\rightsquigarrow_{\beta_v}$ and $\rightsquigarrow_{\beta_i}$ to cover all cases, the environment e' needs to satisfy two properties. First, values are not hidden behind chains of renamings, that is, if $e(x) = y$ then $e(y)$ is not a value. Second, if $e(x)$ is a inert bite, then it reads back to a inert term. These two invariants are nicely expressed in "read-back form" via $\sigma_{e'}$: on any reachable state $e \triangleleft e'$

- e' *is a fireball substitution*, that is, $\sigma_{e'}(x)$ is a fireball for every $x \in \mathsf{dom}(\sigma_{e'})$.
- e' *has immediate values*, that is, if $\sigma_{e'}(x)$ is a value and $x \neq \star$ then $e'(x)$ is a value.

Useful Sharing: The OCAM implements useful sharing because it copies only abstractions and only *on-demand*, that is, only on variable occurrences that are applied, namely on y in the definitions of $\rightsquigarrow_{\beta_v}$ and $\rightsquigarrow_{\beta_i}$.

Overhead Transitions: The overhead transition $\rightsquigarrow_{\mathsf{ren}}$ eliminates explicit renamings, that is, ES containing variables:

$$e[x\leftarrow y] \triangleleft e' \rightsquigarrow_{\mathsf{ren}} e\{x\leftarrow y\} \triangleleft e'$$

when $x \neq \star$. The forthcoming pristine invariant of the machine guarantees that x always has at most one occurrence in e (Lemma VIII.3 below), so that $\rightsquigarrow_{\mathsf{ren}}$ shall not be costly.

The overhead transition $\rightsquigarrow_{\mathsf{sea}}$ simply moves the pointer $\triangleleft$ to the left when no other rule is applicable, i.e. when 1) b is a value, or 2) when b is y or yz but $e'(y)$ is not a value:

$$e[x\leftarrow b] \triangleleft e' \rightsquigarrow_{\mathsf{sea}} e \triangleleft [x\leftarrow b]e'$$

Example VIII.1. *Let us see how the OCAM reduces the environment from Ex. VII.2:*

$$
\begin{array}{l}
[\star\leftarrow \mathtt{xy}][\mathtt{x}\leftarrow\lambda x.[\star\leftarrow \mathtt{zw}][\mathtt{z}\leftarrow\lambda y.[\star\leftarrow y]][\mathtt{w}\leftarrow x\,x]][\mathtt{y}\leftarrow\lambda z.[\star\leftarrow z\,z]] \triangleleft \rightsquigarrow_{\mathsf{sea}}\\
[\star\leftarrow \mathtt{xy}][\mathtt{x}\leftarrow\lambda x.[\star\leftarrow \mathtt{zw}][\mathtt{z}\leftarrow\lambda y.[\star\leftarrow y]][\mathtt{w}\leftarrow x\,x]] \triangleleft [\mathtt{y}\leftarrow\lambda z.[\star\leftarrow z\,z]] \rightsquigarrow_{\mathsf{sea}}\\
[\star\leftarrow \mathtt{xy}] \triangleleft [\mathtt{x}\leftarrow\lambda x.[\star\leftarrow \mathtt{zw}][\mathtt{z}\leftarrow\lambda y.[\star\leftarrow y]][\mathtt{w}\leftarrow x\,x]][\mathtt{y}\leftarrow\lambda z.[\star\leftarrow z\,z]] \rightsquigarrow_{\beta_v}\\
[\star\leftarrow \mathtt{zw}][\mathtt{z}\leftarrow\lambda y.[\star\leftarrow y]][\mathtt{w}\leftarrow \mathtt{y}\,\mathtt{y}] \triangleleft [\mathtt{x}\leftarrow\cdots][\mathtt{y}\leftarrow\lambda z.[\star\leftarrow z\,z]] \rightsquigarrow_{\beta_v}\\
[\star\leftarrow \mathtt{zw}][\mathtt{z}\leftarrow\lambda y.[\star\leftarrow y]][\mathtt{w}\leftarrow \mathtt{y}\,\mathtt{y}] \triangleleft [\mathtt{x}\leftarrow\cdots][\mathtt{y}\leftarrow\lambda z.[\star\leftarrow z\,z]] \rightsquigarrow_{\beta_v} \ldots
\end{array}
$$

The first two steps apply the search transition: the two rightmost ES bind values, thus they can just be skipped since there is nothing to evaluate. The next steps apply a β_λ transition, substituting the body of the abstraction bound in the environment by $\mathtt{y}$*. Evaluation then loops infinitely because the environment has no normal form.*

A. Implementation Theorem

The main property for the implementation theorem is relaxed β-projection. At the open level, one β-transition projects to one $\rightarrow_{\mathsf{om}}$ step, plus one $\rightarrow_{\mathsf{oe}}$ step if the transition is $\rightsquigarrow_{\beta_v}$. To prove it, we need that after read-back 1) the bite yz rewritten by β-transitions becomes a $\rightarrow_{\mathsf{om}}$ redex (with a value as an argument for $\rightsquigarrow_{\beta_v}$) and 2) it occurs in an open context. Becoming a $\rightarrow_{\mathsf{om}}$ redex is guaranteed by the two invariants above. Occurring in an open context instead requires the further invariant below.

Now, if $s = e[x\leftarrow yz] \triangleleft e'$ performs a β-transition, by modular read-back (Lemma VII.6) $s\downarrow = L_{e'}\langle(e[x\leftarrow yz])\downarrow\sigma_{e'}\rangle$. We have that $L_{e'}$ is an open context. Let's focus on $e[x\leftarrow yz]$. The invariant is that the environments on the left of $\triangleleft$ are *pristine*, that is, that in our case e unfolds to $O\langle x\rangle$ for some open context O such that $x \notin \mathsf{vars}(O)$.

Definition VIII.2 (Pristine)**.** Pristine environments *and* pristine bites *are mutually defined as follows:*

- Environments*: ϵ is a pristine environment; $e[x\leftarrow b]$ is pristine if e and b are pristine, $x \in V_{\mathsf{cr}}$, and $e\downarrow = O\langle x\rangle$ for some open context O such that $x \notin \mathsf{vars}(O)$.*
- Bites*: inert bites are pristine; $\lambda x.e$ is pristine if e is pristine.*

The following property shall be crucial for the complexity analysis in Sect. XI.

Lemma VIII.3. *If $e[x\leftarrow b]$ is pristine and well-named and $x \neq \star$, then x occurs exactly once in e.* Proof p. 32

Now, if $s = e[x\leftarrow yz] \triangleleft e'$ the invariants give $s\downarrow = L_{e'}\langle O\langle yz\rangle\sigma_{e'}\rangle = L_{e'}\langle O\sigma_{e'}\langle y\sigma_{e'}\, z\sigma_{e'}\rangle\rangle$ which has the desired shape because $L_{e'}\langle O\sigma_{e'}\rangle$ is open—both O and $L_{e'}$ are open, and open contexts are stable by substitution and plugging—and $y\sigma_{e'}\, z\sigma_{e'}$ is a $\mapsto_{\mathsf{m}}$-redex, as seen before. Structural equivalence $\equiv$ plays a role in the projection of $\rightsquigarrow_{\beta_i}$, to put the created ES $[w\leftarrow z]$ at the right place.

For the halt property, note that final states have the form $\epsilon \triangleleft e'$ that by definition of read-back, Lemma VII.6, and the fireball substitution invariant, read-backs to $L_{e'}\langle f\rangle$, where f is a fireball. A last invariant ensures that the substitution context $L_{e'}$ induced by every reachable state $e \triangleleft e'$ is a *inert (substitution) context* that is, it contains only inert terms—$L_{e'}\langle f\rangle$ is then a fireball, as required.

Spelling out the details, one obtains that the OCAM is a relaxed implementation of $(\to_{\mathsf{o}}, \equiv)$.

IX. The Strong Crumbling Machine

Here we define the Strong Crumbling Abstract Machine (SCAM), building on the previous section.

Basically, we extend the OCAM with a new *strong phase*, identified by a new separator $\triangleright$, whose task is to look for ES $[x{\leftarrow}v]$ containing values and, once one is found, evaluating v under abstraction if x occurs somewhere in the state, and garbage collect it otherwise.

Garbage Collection: Consider the term $t := (\lambda y.x)(\lambda z.\Omega)$ that evaluates in two $\to_{\mathsf{x}}$ steps to x while containing the diverging subterm $\lambda z.\Omega$. The OCAM executed on t produces the final state $s := \triangleleft[\star{\leftarrow}x][y{\leftarrow}\lambda z.\underline{\Omega}]$. Now its strong extension should search for ES containing values in s, but it has to avoid entering $[y{\leftarrow}\lambda z.\underline{\Omega}]$ otherwise it would diverge, breaking the correspondence with $\to_{\mathsf{x}}$. It follows that the machine has to track which ES are garbage and which are not. Let us assume for now that the machine can check it easily; the next section discusses how to implement it.

Deep Implosive Sharing: The machine enters into values whose ES $[x{\leftarrow}v]$ is such that x does occur, potentially *many* times. This ingredient accounts for *deep implosive* sharing. A key point is that the machine enters only inside ES that shall no longer substitute their values (they are left there pending because of useful sharing, as they bind useless occurrences only)—this is why the crucial subterm invariant for complexity analyses (stating that terms duplicated along the whole evaluation with sharing are subterms of the initial term) is not compromised. Useful sharing, instead, is already implemented by the OCAM, and thus simply inherited by the SCAM.

SCAM: First of all, we generalize the right component of states, so as to account for evaluation positions under abstraction. The idea is that the right environment e' can be seen as a context, namely $\langle\cdot\rangle e'$, and that going under abstraction simply requires a further context construction.

$$\begin{array}{rrcl} \text{Machine contexts} & K & ::= & \langle\cdot\rangle e' \mid e[x{\leftarrow}\lambda y.K]e' \\ \text{States} & s & ::= & e \triangleleft K \mid e \triangleright K \end{array}$$

Often $\langle\cdot\rangle\epsilon$ and $e[x{\leftarrow}\lambda y.K]\epsilon$ are noted $\langle\cdot\rangle$ and $e[x{\leftarrow}\lambda y.K]$, respectively. Plugging inside machine contexts, of both environments and machine contexts, is defined as expected, and noted $K\langle e\rangle$ and $K\langle K'\rangle$. We use $\bowtie$ for an unspecified separator, *i.e.* $\bowtie \in \{\triangleleft, \triangleright\}$. The idea is that a state $e \bowtie K$ represent the environment $K\langle e\rangle$ and the active point is between K and e, possibly deep inside many abstractions in $K\langle e\rangle$. Compilation is now defined as $t^\circ := \underline{t} \triangleleft \langle\cdot\rangle$ for a well-named λ-term t, and read-back as $(e \bowtie K)\downarrow := K\langle e\rangle\downarrow$. A state $e \bowtie K$ is well-named if $K\langle e\rangle$ is well-named.

The Open Phase: To lift the transitions of the open machine, we have to define the environment e_K induced by a machine context K, playing the role played by e' in Sect. VIII.

Definition IX.1. *The* environment e_K *of* K *is given by* $e_{\langle\cdot\rangle e'} := e'$ *and* $e_{e[x\leftarrow\lambda y.K]e'} := e_K e'$.

The transitions of the OCAM smoothly lift, as shown in Fig. 2 (where $(*)$ stands for "$(e_K(y))^\alpha = \lambda w.([\star{\leftarrow}b]e')$"). Note that the last transition applies when 1) b is a value, or 2) b is y or yz but $e_K(y)$ is not a value.

The Strong Phase: There are 5 new transitions, that on a state $e \triangleright K$ inspect K rather than e, and accumulate in e the ES that survived the strong phase, which are now fully evaluated. The transitions inspect K from the inside, that is, by looking at what is on the right of the hole $\langle\cdot\rangle$, see Fig. 2.

The union of the nine transitions of the SCAM is noted $\rightsquigarrow_{SCAM}$. Let us explain the new ones.

- $\rightsquigarrow_{\mathsf{sea}_2}$ simply switches to the strong phase, when the current open phase is over.
- $\rightsquigarrow_{\mathsf{sea}_3}$ moves the next ES $[x{\leftarrow}b]$ of the context to the fully evaluated e, if b is not a value.
- $\rightsquigarrow_{\mathsf{gc}}$ garbage collects $[x{\leftarrow}v]$ when x does not occur in e. Checking only e for occurrences is correct: a well-named invariant shall guarantee that x cannot occur in K.
- $\rightsquigarrow_{\mathsf{sea}_4}$ handles the case in which the search has fully processed the current body e of a value, and thus it reunites the enclosing abstraction λy in K with e, adding $\lambda y.e$ to the fully evaluated environment e', and resumes search at the upper abstraction level.
- $\rightsquigarrow_{\mathsf{sea}_5}$ enters into the body e' of the next ES in K, when its content is a value. Searching for values is temporarily over, the machine switches back to the open phase to evaluate e'. The fully evaluated environment e and the enclosing abstraction λy are then moved to K, and the body e' becomes the new left component of the state.

X. The Strong Implementation Theorem

The definition of the SCAM is a smooth generalization of the OCAM, but its implementation theorem is considerably more sophisticated because of deep implosive sharing. We overview the main concepts here, many details are in Appendix H (p. 35). The obstacle is the relaxed β-projection property.

Multi Contexts: The crux is showing that the read back $K\downarrow$ of a machine context (defined below, but keep reading) is an external (evaluation) context E, first of all because... it is not. Both strong and machine contexts have exactly one hole, but if $K = e[x{\leftarrow}\lambda y.K']e'$ then x may have many occurrences in e, causing duplications of the hole via read-back—this is the deep implosive sharing ingredient. The first step, then, is to model $K\downarrow$ as a VSC *multi context*. We need generic multi contexts, plus external and rigid variants.

MULTI CONTEXTS

$$\begin{array}{rrcl} \text{GENERIC} & \mathbb{C} & ::= & \langle\cdot\rangle \mid x \mid \lambda x.\mathbb{C} \mid \mathbb{C}[x\leftarrow\mathbb{C}] \mid \mathbb{C}\mathbb{C} \\ \text{EXTERNAL} & \mathbb{E} & ::= & \langle\cdot\rangle \mid t \mid \lambda x.\mathbb{E} \mid \mathbb{R} \mid \mathbb{E}[x\leftarrow\mathbb{R}] \\ \text{RIGID} & \mathbb{R} & ::= & x \mid \mathbb{R}\mathbb{E} \mid \mathbb{R}[x\leftarrow\mathbb{R}] \end{array}$$

Multi contexts may have no holes, and thus be a term[4]. A multi context is *proper* if it has at least one hole. Plugging $\mathbb{C}\langle t\rangle$ plugs t in all the holes of $\mathbb{C}$, erasing t if $\mathbb{C}$ is not proper.

The next lemma shows that $\mathbb{E}$ is the right generalizations of E: for every external step $t \to_a u$ with $a \in \{\mathsf{xm}, \mathsf{xe}\}$, each replaced hole in $\mathbb{E}\langle t\rangle$ is a $\to_a$ redex, and the firing of the redex gives the expected result $\mathbb{E}\langle u\rangle$—similarly for $\mathbb{R}$, see Appendix H (p. 35). We avoid on purpose the definition of a parallel step, that would induce a more complex notion of implementation—parallelism here is caught more flexibly by the diamond property of $\to_{\mathsf{x}}$. For technical reasons, we prove a more general result:

Proof p. 36 **Lemma X.1** (Multi step)**.** *Let $\mathbb{E}$ be a proper external multi context with k holes and $\{a_1, \dots, a_n\} \subseteq \{\mathsf{xm}, \mathsf{xe}\}$.*
If $t \to_{a_1} \cdots \to_{a_n} u$ then $\mathbb{E}\langle t\rangle\,(\to_{a_1} \cdots \to_{a_n})^k\,\mathbb{E}\langle u\rangle$ where the i-th sequence of steps has the shape $E_i\langle t\rangle \to_{a_1} \cdots \to_{a_n} E_i\langle u\rangle$ for an external context E_i, for every $i \in \{1, \dots, k\}$.

Read back then extends to K via the following clauses (reading back ES without the hole as before), obtaining a multi context: $\langle\cdot\rangle{\downarrow} := \langle\cdot\rangle$ and $(e[x\leftarrow\lambda y.K]){\downarrow} := e{\downarrow}\{x\leftarrow\lambda y.K{\downarrow}\}$. Moreover, it factors in a way similar to the open case (Lemma VII.6).

Proof p. 37 **Lemma X.2** (Modular read-back)**.** $K\langle e\rangle{\downarrow} = K{\downarrow}\langle e{\downarrow}\sigma_{e_K}\rangle$.

Invariants: Let K be the machine context of a reachable state s. Proving that $K{\downarrow}$ is a proper external multi context $\mathbb{E}$ requires delicate and involved invariants, that build on those for the OCAM. An essential concept is the *frame* of K, that isolates its fully evaluated part plus the hole.

$$\text{FRAMES} \qquad F \quad ::= \quad \langle\cdot\rangle \mid e[x\leftarrow\lambda y.F]$$

The *frame* F_K *of a context* K is defined by $F_{\langle\cdot\rangle e'} := \langle\cdot\rangle$ and $F_{e[x\leftarrow\lambda y.K]e'} := e[x\leftarrow\lambda y.F_K]$.

Proving that $K{\downarrow}$ is *proper*, requires an invariant ensuring that F_K is *garbage-free*, so that every variable bound by an ES (but $\star$) occurs, implying that $(e[x\leftarrow\lambda y.K]){\downarrow} = e{\downarrow}\{x\leftarrow\lambda y.K{\downarrow}\}$ does not erase $\lambda y.K{\downarrow}$ and that $K{\downarrow}$ is itself proper.

Proving that $K{\downarrow}$ is *external*, requires a sophisticated *goodness* invariant, building on the invariants of the OCAM. Roughly, $K{\downarrow}$ is given by $F_K{\downarrow}$ where one applies the substitution σ_{e_K} and shuffles around the ES in L_{e_K}. Oversimplifying, goodness says that $F_K{\downarrow}$ is an external multi context that stays so also after the application of σ_{e_K} and the shuffling of L_{e_K}.

Proof p. 51 **Theorem X.3** (Contextual read-back)**.** *Let $s = e \bowtie K$ be a reachable state. Then $K{\downarrow}$ is a proper external multi context.*

[4]It is easily seen that the grammar of $\mathbb{C}$ allows to generate all terms, and the one for $\mathbb{R}$ all rigid terms—see Lemma H.2 in the Appendix (page 35). For $\mathbb{E}$, terms are simply injected in by the grammar itself.

Obtaining Thm. X.3 is the difficult and involved step. Then, β-transitions are smoothly projected on $\to_{\mathsf{x}}$ steps, and the implementation theorem easily follows.

Theorem X.4 (SCAM implementation)**.**

1) Relaxed β-projection*: let s be a reachable state. If $s \rightsquigarrow_{\beta_v} s'$ then $s{\downarrow}(\to_{\mathsf{xm}}\to_{\mathsf{xe}})^+ s'{\downarrow}$, and if $s \rightsquigarrow_{\beta_i} s'$ then $s{\downarrow} \to^+_{\mathsf{xm}} \equiv s'{\downarrow}$.* Proof p. 52 Proof p. 52
2) Strong implementation*: the SCAM is a relaxed implementation of the external strategy $(\to_{\mathsf{x}}, \equiv)$.* Proof p. 54

XI. COMPLEXITY ANALYSIS OF THE SCAM

Here we prove that the SCAM can be implemented within a bilinear overhead. The proof is simple and mostly follows a standard schema: we bound the number of overhead steps, then bound the cost of single steps, and end by combining the two. It all rests on the *size invariant* below, that is a quantitative form of the subterm invariant needed for all complexity analyses of abstract machines.

The size $|t|$ of the initial term t is one of the two parameters of the analysis, linearly preserved by compilation $t^\circ = \underline{t} \triangleleft \langle\cdot\rangle$.

λ-TERMS	BITES	ENVS
$\lvert x\rvert := 1$	$\lvert x\rvert := 1$	$\lvert\epsilon\rvert := 0$
$\lvert tu\rvert := \lvert t\rvert + \lvert u\rvert + 1$	$\lvert xy\rvert := 1$	$\lvert e[x\leftarrow b]\rvert :=$
$\lvert\lambda x.t\rvert := \lvert t\rvert + 1$	$\lvert\lambda x.e\rvert := 1 + \lvert e\rvert$	$1 + \lvert e\rvert + \lvert b\rvert$

Lemma XI.1 (Linear compilation)**.** *Let t be a λ-term. Then $|\underline{t}| \le 2|t|$.* Proof p. 54

The size invariant provides a size bound on the values that may be duplicated by β-transitions, which are the only ones making the size of states grow.

Lemma XI.2 (Size invariant)**.** *Let $s = e \bowtie K$ be a state reachable from $s_0 = e_0 \triangleleft \langle\cdot\rangle$. Then $|v| \le |e_0|$ for every value v that occurs in e or e_K if $\bowtie = \triangleleft$, or in e_K if $\bowtie = \triangleright$.* Proof p. 55

Number of Overhead Transitions: We bound the number of overhead transitions in a modular way with respect to the two phases of the machine. First, a global analysis shows that the number of transitions of all strong phases is bounded by the number of transitions of all open phases:

Lemma XI.3 (Open phases bound strong phases)**.** *Let $\rho\colon s_0 \rightsquigarrow s$ an execution of the SCAM. Then: $|\rho|_{\mathsf{sea}_2} + |\rho|_{\mathsf{sea}_3} + |\rho|_{\mathsf{gc}} + |\rho|_{\mathsf{sea}_4} + |\rho|_{\mathsf{sea}_5} \le |\rho|_{\beta_i} + 4|\rho|_{\mathsf{sea}_1} + 1$.* Proof p. 56

To quantify the overhead of the open phase, we introduce a new measure $\|\cdot\|$ over machine states, tailored to decrease on all open transitions but βs.

CONTEXTS	STATES
$\|\langle\cdot\rangle\| := 0$	$\|e \triangleleft K\| := \lvert e\rvert + \|K\|$
$\|e[x\leftarrow\lambda y.K]\| := \|K\|$	$\|e \triangleright K\| := \|K\|$
$\|K[x\leftarrow b]\| := \|K\| + \lvert b\rvert$	

Lemma XI.4 (Measure during execution)**.** *Let s be a state reachable from s_0, and $s \rightsquigarrow_a s'$.* Proof p. 56

1) *Beta transitions increase the measure: if $a \in \{\beta_\lambda, \beta_i\}$ then $\|s'\| \le \|s\| + \|s_0\|$.*

2) *Open overhead decreases the measure: if* $a \in \{\mathsf{ren}, \mathsf{sea}_1\}$ *then* $\|s'\| < \|s\|$.
3) *Strong phase does not increase the measure: if* $a \in \{\mathsf{sea}_2, \mathsf{sea}_3, \mathsf{gc}, \mathsf{sea}_4, \mathsf{sea}_5\}$ *then* $\|s'\| \leq \|s\|$.

By combining the two previous lemmas, we obtain a bilinear bound on the total overhead:

Corollary XI.5 (Bilinear number of overhead transitions)**.** *Let* t *be a* λ*-term and* $\rho \colon t^\circ \rightsquigarrow^*_{SCAM} s$ *be a SCAM execution. Then* $|\rho| \in \mathcal{O}((1 + |\rho|_\beta)|t|)$.

Cost of Single Steps: We need high-level assumptions on how bites and crumbled environments are concretely implemented—a reference implementation in OCaml is explained in Appendix J, p. 59 and can be downloaded at https://github.com/sacerdot/SCAM.

As it is standard for machines with global environments (see Accattoli and Barras [5]), variables are represented as memory locations, variable occurrences as pointers, and an ES $[x \leftarrow b]$ is the fact that the location associated with x contains b—this allows $\mathcal{O}(1)$ look-up in environments. The copy/renaming in β-transitions costs $\mathcal{O}(|t|)$ by the size invariant, following [5], essentially implementing the proof-nets representation of a term, that can be seen as a pointer-based DAG.

The SCAM visits the proof-net/DAG in bi-directional ways: environments are visited both right-to-left (open phases) and left-to-right (strong phases), and abstractions are entered/exited at phase switches. Rather than having doubly linked nodes we adapt to our more general DAG framework the subtler space-conscious technique by McBride [47], itself generalizing the standard zipper technique for lists by Huet [42], obtaining bi-directional moves over the graph in $\mathcal{O}(1)$.

To implement $\rightsquigarrow_{\mathsf{ren}}$ in $\mathcal{O}(1)$ we need to jump to the occurrence of the renamed variable. By Lemma VIII.3, the renamed variable has at most one occurrence when the rule fires. An implementation can thus keep with no overhead a bi-directional link between the variable and its occurrence — as long as the variable occurs once.

For $\rightsquigarrow_{\mathsf{gc}}$, variables also carry a reference counter, to test if they occur in $\mathcal{O}(1)$. By the size invariant, the size of erased values is $\mathcal{O}(|t|)$, but since there can be $\mathcal{O}((1 + |\rho|_\beta)|t|) \rightsquigarrow_{\mathsf{gc}}$ steps (Corollary XI.5), we obtain a bound quadratic in $|t|$, which is too loose. The stricter bilinear bound is inferred via global analysis. Indeed, by the size invariant the size of a state is bounded by $\mathcal{O}((1 + |\rho|_\beta)|t|)$. So, the global cost of erasing cannot be more than $\mathcal{O}((1 + |\rho|_\beta)|t|)$. Summing it all up,

Proof p. 58

Theorem XI.6 (The SCAM is bilinear)**.** *Let* t *be a* λ*-term and* $\rho \colon t^\circ \rightsquigarrow^*_{SCAM} s$ *a SCAM execution. Then* ρ *can be implemented on a RAM in* $\mathcal{O}((1 + |\rho|_\beta)|t|)$.

Via the implementation theorem (Theorem X.4), we obtain reasonable cost models for Strong CbV.

Corollary XI.7 (Reasonable cost models)**.** *Let* t *be a* λ*-term,* $d : t \rightarrow^*_{\mathsf{x}} u$*, and* $\rho \colon t^\circ \rightsquigarrow^*_{SCAM} s$*. Then* $|d|_{\mathsf{m}}$ *and* $|\rho|_{\mathsf{m}}$ *are reasonable time cost models for Strong CbV.*

XII. Implosiveness at Work

Let us give an example of the implosive phenomenon. Consider the following families of terms $t_1 \coloneqq \pi I$ and $t_{n+1} \coloneqq \pi(\lambda z.t_n)$, where $\pi \coloneqq \lambda x.\lambda y.((yx)x)$, and $u_1 \coloneqq \lambda y.((yI)I)$ and $u_{n+1} \coloneqq \lambda y.((y(\lambda z.u_n))(\lambda z.u_n))$.

An easy induction shows that $\rightarrow_{\mathsf{x}}$ takes on t_n a number of steps exponential in n. Note that 1) all the β-redexes of t_n are given by a copy of π with just one argument, which is a value, 2) the argument is duplicated, and that 3) these facts are stable by evaluation. Therefore, substitution always happens on occurrences of x as arguments. The SCAM never substitutes on arguments, and evaluates them while they are shared, thus avoiding the exponential duplication of redexes.

Proof p. 58

Proposition XII.1 (Implosive family)**.** *Let* t_n *and* u_n *as above.*

Proof p. 63

1) External strategy (exponentially many steps)*:* $t_n(\rightarrow_{\mathsf{xm}}\rightarrow_{\mathsf{xe}})^{2^n-1}u_n$ *and* u_n *is a strong fireball.*
2) SCAM Implosion (linearly many steps)*:* $\rho_n : t_n^\circ \rightsquigarrow^*_{SCAM} s_n$ *with* $s_n{\downarrow} = u_n$ *and* $|\rho_n|_\beta = n$.

Acknowledgements. This work has been partially funded by the ANR JCJC grant 'COCA HOLA' (ANR-16-CE40-004-01); the third author by the MIUR-PRIN project 'Analysis of Program Analyses' (ASPRA, ID 201784YSZ5 004) and the Italian INdAM – GNCS project 2020 'Reversible Concurrent Systems: from Models to Languages'.

References

[1] B. Accattoli, "Proof nets and the call-by-value λ-calculus," *Theor. Comput. Sci.*, vol. 606, pp. 2–24, 2015.
[2] ——, "The complexity of abstract machines," in *WPTE@FSCD*, ser. EPTCS, vol. 235, 2016, pp. 1–15.
[3] ——, "The useful MAM, a reasonable implementation of the strong λ-calculus," in *Logic, Language, Information, and Computation - 23rd International Workshop, WoLLIC 2016, Puebla, Mexico, August 16-19th, 2016. Proceedings*, 2016, pp. 1–21.
[4] B. Accattoli, P. Barenbaum, and D. Mazza, "Distilling abstract machines," in *19th ACM SIGPLAN International Conference on Functional Programming, ICFP 2014*. ACM, 2014, pp. 363–376.
[5] B. Accattoli and B. Barras, "Environments and the complexity of abstract machines," in *19th International Symposium on Principles and Practice of Declarative Programming, PPDP 2017*. ACM, 2017, pp. 4–16.
[6] B. Accattoli, A. Condoluci, G. Guerrieri, and C. Sacerdoti Coen, "Crumbling abstract machines," in *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, 2019, pp. 4:1–4:15.
[7] B. Accattoli and U. Dal Lago, "(Leftmost-outermost) Beta reduction is invariant, indeed," *Logical Methods in Computer Science*, vol. 12, no. 1, 2016.
[8] B. Accattoli and G. Guerrieri, "Open Call-by-Value," in *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016*, ser. Lecture Notes in Computer Science, vol. 10017. Springer, 2016, pp. 206–226.
[9] ——, "Abstract machines for open call-by-value," *Sci. Comput. Program.*, vol. 184, 2019.
[10] B. Accattoli, G. Guerrieri, and M. Leberle, "Semantic bounds and strong call-by-value normalization," *CoRR*, vol. abs/2104.13979, 2021.
[11] B. Accattoli and L. Paolini, "Call-by-value solvability, revisited," in *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*, 2012, pp. 4–16.
[12] B. Accattoli and C. Sacerdoti Coen, "On the relative usefulness of fireballs," in *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, 2015, pp. 141–155.

[13] B. Accattoli and C. Sacerdoti Coen, "On the value of variables," *Information and Computation*, vol. 255, pp. 224–242, 2017. [Online]. Available: https://doi.org/10.1016/j.ic.2017.01.003

[14] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard, "A functional correspondence between evaluators and abstract machines," in *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, 2003, pp. 8–19.

[15] Z. M. Ariola and M. Felleisen, "The Call-By-Need lambda Calculus," *J. Funct. Program.*, vol. 7, no. 3, pp. 265–301, 1997.

[16] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler, "The call-by-need lambda calculus," in *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*. ACM Press, 1995, pp. 233–246.

[17] A. Asperti and S. Guerrini, *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1998.

[18] A. Asperti and H. G. Mairson, "Parallel beta reduction is not elementary recursive," *Inf. Comput.*, vol. 170, no. 1, pp. 49–80, 2001. [Online]. Available: https://doi.org/10.1006/inco.2001.2869

[19] T. Balabonski, P. Barenbaum, E. Bonelli, and D. Kesner, "Foundations of strong call by need," *PACMPL*, vol. 1, no. ICFP, pp. 20:1–20:29, 2017. [Online]. Available: https://doi.org/10.1145/3110264

[20] H. P. Barendregt, *The Lambda Calculus – Its Syntax and Semantics*. North-Holland, 1984, vol. 103.

[21] B. Barras, "Auto-validation d'un système de preuves avec familles inductives," Ph.D. dissertation, Université Paris 7, 1999.

[22] M. Biernacka, D. Biernacki, W. Charatonik, and T. Drab, "An abstract machine for strong call by value," in *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*, 2020, pp. 147–166.

[23] M. Biernacka and W. Charatonik, "Deriving an abstract machine for strong call by need," in *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, 2019, pp. 8:1–8:20.

[24] M. Biernacka, W. Charatonik, and T. Drab, "Strong call by value is reasonable for time," *CoRR*, vol. abs/2102.05985, 2021.

[25] M. Biernacka, W. Charatonik, and K. Zielinska, "Generalized refocusing: From hybrid strategies to abstract machines," in *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, 2017, pp. 10:1–10:17.

[26] G. E. Blelloch and J. Greiner, "Parallelism in sequential functional languages," in *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, 1995, pp. 226–237.

[27] A. Carraro and G. Guerrieri, "A semantical and operational account of call-by-value solvability," in *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, 2014, pp. 103–118.

[28] A. Condoluci, B. Accattoli, and C. S. Coen, "Sharing equality is linear," in *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, 2019, pp. 9:1–9:14. [Online]. Available: https://doi.org/10.1145/3354166.3354174

[29] P. Crégut, "Strongly reducing variants of the Krivine abstract machine," *High. Order Symb. Comput.*, vol. 20, no. 3, pp. 209–230, 2007.

[30] P. Curien and H. Herbelin, "The duality of computation," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, 2000, pp. 233–243.

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

[32] U. Dal Lago and S. Martini, "Derivational complexity is an invariant cost model," in *Foundational and Practical Aspects of Resource Analysis - First International Workshop, FOPARA 2009, Eindhoven, The Netherlands, November 6, 2009, Revised Selected Papers*, 2009, pp. 100–113.

[33] U. Dal Lago and S. Martini, "On Constructor Rewrite Systems and the Lambda Calculus," *Logical Methods in Computer Science*, vol. 8, no. 3, 2012.

[34] O. Danvy and L. R. Nielsen, "Refocusing in Reduction Semantics," BRICS, Tech. Rep. RS-04-26, 2004.

[35] O. Danvy and I. Zerny, "A synthetic operational account of call-by-need evaluation," in *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, 2013, pp. 97–108.

[36] D. de Carvalho, "Execution time of λ-terms via denotational semantics and intersection types," *Math. Str. in Comput. Sci.*, vol. 28, no. 7, pp. 1169–1203, 2018.

[37] T. Ehrhard and L. Regnier, "Böhm trees, Krivine's machine and the Taylor expansion of lambda-terms," in *Logical Approaches to Computational Barriers, Second Conference on Computability in Europe, CiE 2006, Swansea, UK, June 30-July 5, 2006, Proceedings*, 2006, pp. 186–197.

[38] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "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, 1993, pp. 502–514. [Online]. Available: https://doi.org/10.1145/989393.989443

[39] Y. Forster, F. Kunze, and M. Roth, "The weak call-by-value λ-calculus is reasonable for both time and space," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 27:1–27:23, 2020.

[40] Á. García-Pérez and P. Nogueira, "The full-reducing krivine abstract machine KN simulates pure normal-order reduction in lockstep: A proof via corresponding calculus," *J. Funct. Program.*, vol. 29, p. e7, 2019.

[41] B. Grégoire and X. Leroy, "A compiled implementation of strong reduction," in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ICFP '02*. ACM, 2002, pp. 235–246.

[42] G. P. Huet, "The zipper," *J. Funct. Program.*, vol. 7, no. 5, pp. 549–554, 1997. [Online]. Available: http://journals.cambridge.org/action/displayAbstract?aid=44121

[43] D. Kesner, "Reasoning about call-by-need by means of types," in *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, 2016, pp. 424–441.

[44] X. Leroy, "The ZINC experiment: an economical implementation of the ML language," INRIA, Technical report 117, 1990. [Online]. Available: http://gallium.inria.fr/~xleroy/publi/ZINC.pdf

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

[46] J. Maraist, M. Odersky, and P. Wadler, "The Call-by-Need Lambda Calculus," *Journal of Functional Programming*, vol. 8, no. 3, pp. 275–317, 1998.

[47] C. Mcbride, "The derivative of a regular type is its type of one-hole contexts (extended abstract)," 2001.

[48] L. Paolini, "Call-by-value separability and computability," in *Theoretical Computer Science, 7th Italian Conference, ICTCS 2001, Torino, Italy, October 4-6, 2001, Proceedings*, 2001, pp. 74–89.

[49] L. Paolini and S. Ronchi Della Rocca, "Call-by-value solvability," *RAIRO Theor. Informatics Appl.*, vol. 33, no. 6, pp. 507–534, 1999.

[50] G. D. Plotkin, "Call-by-Name, Call-by-Value and the lambda-Calculus," *Theoretical Computer Science*, vol. 1, no. 2, pp. 125–159, 1975.

[51] S. Ronchi Della Rocca and L. Paolini, *The Parametric λ-Calculus – A Metamodel for Computation*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[52] A. Sabry and M. Felleisen, "Reasoning about Programs in Continuation-Passing Style," *Lisp and Symbolic Computation*, vol. 6, no. 3-4, pp. 289–360, 1993.

[53] C. Sacerdoti Coen, "Reduction and Conversion Strategies for the Calculus of (co)Inductive Constructions: Part I," *Electronic Notes in Theoretical Computer Science*, vol. 174, pp. 97–118, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S157106610700401X

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

[55] C. F. Slot and P. van Emde Boas, "On tape versus core; an application of space efficient perfect hash functions to the invariance of space," in *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*, 1984, pp. 391–400.

[56] C. P. Wadsworth, "Semantics and pragmatics of the lambda-calculus," PhD Thesis, Oxford, 1971, chapter 4.

TECHNICAL APPENDIX

APPENDIX A
PRELIMINARIES AND NOTATIONS IN REWRITING

For a relation R on a set of terms, R^* is its reflexive-transitive closure. Given a relation $\to_{\mathsf{r}}$, an r-*evaluation* (or simply evaluation if unambiguous) d is a finite sequence of terms $(t_i)_{0\le i\le n}$ (for some $n \ge 0$) such that $t_i \to_{\mathsf{r}} t_{i+1}$ for all $1 \le i < n$, and we write $d\colon t \to_{\mathsf{r}}^* u$ if $t_0 = t$ and $t_n = u$. The *length* n of d is denoted by $|d|$, and $|d|_a$ is the number of a-*steps* (*i.e.* the number of $t_i \to_a t_{i+1}$ for some $1 \le i \le n$) in d, for a given subrelation $\to_a$ of $\to_{\mathsf{r}}$.

A term t is r-*normal* if there is no u such that $t \to_{\mathsf{r}} u$. An evaluation $d\colon t \to_{\mathsf{r}}^* u$ is r-*normalizing* if u is r-normal. A term t is *weakly* r-*normalizing* if there is a r-normalizing evaluation $d\colon t \to_{\mathsf{r}}^* u$; and t is *strongly* r-*normalizing* if there no infinite sequence $(t_i)_{i\in\mathbb{N}}$ such that $t_0 = t$ and $t_i \to_{\mathsf{r}} t_{i+1}$ for all $i \in \mathbb{N}$. Clearly, strong r-normalization implies weak r-normalization.

A relation $\to_{\mathsf{r}}$ is *diamond* if $u_1 \,{}_{\mathsf{r}}\!\leftarrow t \to_{\mathsf{r}} u_2$ and $u_1 \neq u_2$ imply $u_1 \to_{\mathsf{r}} p \,{}_{\mathsf{r}}\!\leftarrow u_2$ for some p. As a consequence:

1) $\to_{\mathsf{r}}$ is confluent (*i.e.* $u_1 \,{}_{\mathsf{r}}^*\!\leftarrow t \to_{\mathsf{r}}^* u_2$ implies $u_1 \to_{\mathsf{r}}^* p \,{}_{\mathsf{r}}^*\!\leftarrow u_2$ for some p);
2) any term t has at most one normal form (*i.e.* if $t \to_{\mathsf{r}}^* u$ and $t \to_{\mathsf{r}}^* p$ with u and p r-normal, then $u = p$);
3) all r-evaluations with the same start and end terms have the same length (*i.e.* if $d\colon t \to_{\mathsf{r}}^* u$ and $d'\colon t \to_{\mathsf{r}}^* u$ then $|d| = |d'|$);
4) t is weakly r-normalizing iff it is strongly r-normalizing.

Two relations $\to_{\mathsf{r}_1}$ and $\to_{\mathsf{r}_2}$ *strongly commute* if $u_1 \,{}_{\mathsf{r}_1}\!\leftarrow t \to_{\mathsf{r}_2} u_2$ implies $u_1 \to_{\mathsf{r}_2} p \,{}_{\mathsf{r}_2}\!\leftarrow u_2$ for some p. If $\to_{\mathsf{r}_1}$ and $\to_{\mathsf{r}_2}$ strongly commute and are diamond, then

1) $\to_{\mathsf{r}} = \to_{\mathsf{r}_1} \cup \to_{\mathsf{r}_2}$ is diamond,
2) all r-evaluations with the same start and end terms have the same number of any kind of steps (*i.e.* if $d\colon t \to_{\mathsf{r}}^* u$ and $d'\colon t \to_{\mathsf{r}}^* u$ then $|d|_{\mathsf{r}_1} = |d'|_{\mathsf{r}_1}$ and $|d|_{\mathsf{r}_2} = |d'|_{\mathsf{r}_2}$).

It is a strong form of confluence and implies *uniform normalization* (if there is a normalizing sequence from t then there are no diverging sequences from t) and the *random descent property* (all normalizing sequences from t have the same length)

APPENDIX B
PROOFS OF SECTION II (VSC)

Lemma B.1 (Shape of strong fireballs)**.** *Let t be a strong fireball. Then exactly one of the following holds:*

- *either t is a strong inert term,*
- *or t is a value fireball.*

Proof. Proving that at least one of the two holds is left for the reader. We now prove that only one of them holds:

- Let t be a strong inert term. We prove that t is not a value fireball by structural induction on t:
  - *Variable*: Trivial.
  - *Application*: Trivial.
  - ES; *i.e.*, $t = i_{\mathsf{s}}[x \leftarrow i'_{\mathsf{s}}]$: Then i_{s} is not a value fireball —by *i.h.*—, and so neither is t.
- Let $t = L\langle \lambda x.f_{\mathsf{s}}\rangle$, with $L = [x_1 \leftarrow i_{s,1}] \dots [x_n \leftarrow i_{s,n}]$, with $n \ge 0$. We prove that t is not a strong inert term by induction on n:

– *Empty substitution context*; *i.e.*, $L = \langle\cdot\rangle$: Trivial.
– *Non-empty substitution context*; *i.e.*, $L = L'[x{\leftarrow}i_{s,n+1}]$: Since $L'\langle\lambda x.f\rangle$ is not a strong inert term —by *i.h.*—, then neither is $L'\langle\lambda x.f\rangle[x{\leftarrow}i_{s,n+1}] = t$. □

Proposition B.2 (Characterization of normal forms)**.** *Let t be a VSC term. t is $\to_{\mathsf{vsc}}$-normal if and only if t is a strong fireball.*

See p. 4 Lemma II.5

Proof. We prove the two directions of the equivalence separately.

1) Let t be vsc-normal. We shall prove that t is a strong fireball by induction on t:
   - *Variable*. Trivial.
   - *Abstraction*; *i.e.*, $t = \lambda x.u$. As t is $\to_{\mathsf{vsc}}$-normal, so is u. By *i.h.*, u is a strong fireball, and then so is t.
   - *Application*; *i.e.*, $t = t_1t_2$. Since t is $\to_{\mathsf{vsc}}$-normal, so are t_1 and t_2. By *i.h.*, t_1 and t_2 are strong fireballs. Note that $t_1 \neq L\langle\lambda x.u\rangle$, otherwise $t \mapsto_{\mathsf{m}} L\langle u[x{\leftarrow}t_2]\rangle$ which contradicts vsc-normality of t. So, t_1 is a strong inert term (by Lemma B.1), thus t is a strong fireball.
   - *Explicit substitution*; *i.e.*, $t = t_1[x{\leftarrow}t_2]$. Since t is vsc-normal, then so are t_1 and t_2. By *i.h.*, t_1 and t_2 are strong fireballs. Note that $t_2 \neq L\langle v\rangle$, otherwise $t \mapsto_{\mathsf{e}} L\langle t_1\{x{\leftarrow}v\}\rangle$ which contradicts the vsc-normality of t. Thus t_2 is a strong inert term (by Lemma B.1), and so t is a strong fireball.
2) Let t be a strong fireball. We prove that t is vsc-normal by induction on the definition of strong fireball.
   - *Variable*. Trivial.
   - *Abstraction*; *i.e.*, $t = \lambda x.f_{\mathsf{s}}$. By *i.h.*, f_{s} is vsc-normal, and hence so is t.
   - *Application*; *i.e.*, $t = i_{\mathsf{s}}f_{\mathsf{s}}$. By *i.h.*, i_{s} and f_{s} are vsc-normal. Since i_{s} is not of the of the form $L\langle\lambda x.u\rangle$ (by Lemma B.1), then t is also vsc-normal.
   - *Explicit substitution*; *i.e.*, $t = f_{\mathsf{s}}[x{\leftarrow}i_{\mathsf{s}}]$ (it includes the case when f_{s} is a strong inert term). By *i.h.*, both f_{s} and i_{s} are vsc-normal. Since i_{s} is not of the form $L\langle v\rangle$ (by Lemma B.1), then t is also vsc-normal. □

Lemma B.3 (Basic Properties of λ_{vsc})**.**

1) $\to_{\mathsf{m}}$ and $\to_{\mathsf{e}}$ are strongly normalizing (separately).
2) $\to_{\mathsf{om}}$ and $\to_{\mathsf{oe}}$ are diamond (separately).
3) $\to_{\mathsf{om}}$ and $\to_{\mathsf{oe}}$ strongly commute.

Proof. The statements of Lemma B.3 are a refinement of some results proved in [11], where $\to_{\mathsf{o}}$ is denoted by $\to_{\mathsf{w}}$.

1) See [11, Lemma 3].
2) We prove that $\to_{\mathsf{om}}$ is diamond, *i.e.* if $u\ {}_{\mathsf{om}}{\leftarrow}\ t \to_{\mathsf{om}} p$ with $u \neq p$ then there exists $t' \in \Lambda_{\mathsf{vsc}}$ such that $u \to_{\mathsf{om}} t'\ {}_{\mathsf{om}}{\leftarrow}\ p$. The proof is by induction on the definition of $\to_{\mathsf{om}}$. Since there $t \to_{\mathsf{om}} p \neq u$ and the reduction $\to_{\mathsf{om}}$ is weak, there are only eight cases:
   - *Step at the Root for $t \to_{\mathsf{om}} u$ and Application Right for $t \to_{\mathsf{om}} p$, i.e. $t := L\langle\lambda x.r\rangle q \mapsto_{\mathsf{m}} L\langle r[x{\leftarrow}q]\rangle =: u$* and $t \mapsto_{\mathsf{m}} L\langle\lambda x.r\rangle q' =: p$ with $q \to_{\mathsf{om}} q'$: then, $u \to_{\mathsf{om}} L\langle r[x{\leftarrow}q']\rangle\ {}_{\mathsf{om}}{\leftarrow}\ p$;
   - *Step at the Root for $t \to_{\mathsf{om}} u$ and Application Left for $t \to_{\mathsf{om}} p$, i.e.*, for some $n > 0$,
   $$t := (\lambda x.r)[x_1{\leftarrow}t_1]\dots[x_n{\leftarrow}t_n]q \mapsto_{\mathsf{m}} r[x{\leftarrow}q][x_1{\leftarrow}t_1]\dots[x_n{\leftarrow}t_n] =: u$$
   whereas $t \to_{\mathsf{om}} (\lambda x.r)[x_1{\leftarrow}t_1]\dots[x_j{\leftarrow}t'_j]\dots[x_n{\leftarrow}t_n]q =: p$ with $t_j \to_{\mathsf{om}} t'_j$ for some $1 \leq j \leq n$: then,
   $$u \to_{\mathsf{om}} r[x{\leftarrow}q][x_1{\leftarrow}t_1]\dots[x_j{\leftarrow}t'_j]\dots[x_n{\leftarrow}t_n]\ {}_{\mathsf{om}}{\leftarrow}\ p;$$
   - *Application Left for $t \to_{\mathsf{om}} u$ and Application Right for $t \to_{\mathsf{om}} p$, i.e. $t := qr \to_{\mathsf{om}} q'r =: u$* and $t \to_{\mathsf{om}} qr' =: p$ with $q \to_{\mathsf{om}} q'$ and $r \to_{\mathsf{om}} r'$: then, $u \to_{\mathsf{om}} q'r'\ {}_{\mathsf{om}}{\leftarrow}\ p$;
   - *Application Left for both $t \to_{\mathsf{om}} u$ and $t \to_{\mathsf{om}} p$, i.e. $t := qr \to_{\mathsf{om}} q'r =: u$* and $t \to_{\mathsf{om}} q''r =: p$ with $q'\ {}_{\mathsf{om}}{\leftarrow}\ q \to_{\mathsf{om}} q''$: by *i.h.*, there exists $q_0 \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{om}} q_0\ {}_{\mathsf{m}}{\leftarrow}\ q''$, hence $u \to_{\mathsf{om}} q_0r\ {}_{\mathsf{m}}{\leftarrow}\ p$;
   - *Application Right for both $t \to_{\mathsf{om}} u$ and $t \to_{\mathsf{om}} p$, i.e. $t := rq \to_{\mathsf{om}} rq' =: u$* and $t \to_{\mathsf{om}} rq'' =: p$ with $q'\ {}_{\mathsf{om}}{\leftarrow}\ q \to_{\mathsf{om}} q''$: by *i.h.*, there exists $q_0 \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{om}} q_0\ {}_{\mathsf{om}}{\leftarrow}\ q''$, hence $u \to_{\mathsf{om}} rq_0\ {}_{\mathsf{om}}{\leftarrow}\ p$;
   - ES *left for $t \to_{\mathsf{om}} u$ and* ES *right for $t \to_{\mathsf{om}} p$, i.e. $t := q[x{\leftarrow}r] \to_{\mathsf{om}} q'[x{\leftarrow}r] =: u$* and $t \to_{\mathsf{om}} q[x{\leftarrow}r'] =: p$ with $q \to_{\mathsf{om}} q'$ and $r \to_{\mathsf{om}} r'$: then,
   $$u \to_{\mathsf{om}} q'[x{\leftarrow}r']\ {}_{\mathsf{om}}{\leftarrow}\ p$$
   - ES *left for both $t \to_{\mathsf{om}} u$ and $t \to_{\mathsf{om}} p$, i.e. $t := q[x{\leftarrow}r] \to_{\mathsf{om}} q'[x{\leftarrow}r] =: u$* and $t \to_{\mathsf{om}} q''[x{\leftarrow}r] =: p$ with $q'\ {}_{\mathsf{om}}{\leftarrow}\ q \to_{\mathsf{om}} q''$: by *i.h.*, there exists $q_0 \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{om}} q_0\ {}_{\mathsf{om}}{\leftarrow}\ q''$, hence $u \to_{\mathsf{m}} q_0[x{\leftarrow}r]\ {}_{\mathsf{om}}{\leftarrow}\ p$;
   - ES *right for both $t \to_{\mathsf{om}} u$ and $t \to_{\mathsf{om}} p$, i.e. $t := r[x{\leftarrow}q] \to_{\mathsf{om}} r[x{\leftarrow}q'] =: u$* and $t \to_{\mathsf{om}} r[x{\leftarrow}q''] =: p$ with $q'\ {}_{\mathsf{om}}{\leftarrow}\ q \to_{\mathsf{m}} q''$: by *i.h.*, there exists $q_0 \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{om}} q_0\ {}_{\mathsf{om}}{\leftarrow}\ q''$, hence $u \to_{\mathsf{m}} r[x{\leftarrow}q_0]\ {}_{\mathsf{om}}{\leftarrow}\ p$.

We prove that $\to_{\mathsf{oe}}$ is diamond, *i.e.* if $u \ {}_{\mathsf{oe}}\!\leftarrow t \to_{\mathsf{oe}} p$ with $u \neq p$ then there exists $q \in \Lambda_{\mathsf{vsc}}$ such that $u \to_{\mathsf{oe}} t' \ {}_{\mathsf{e}}\!\leftarrow p$. The proof is by induction on the definition of $\to_{\mathsf{oe}}$. Since there $t \to_{\mathsf{oe}} p \neq u$ and the reduction $\to_{\mathsf{oe}}$ is weak, there are only eight cases:

- *Step at the Root for* $t \to_{\mathsf{oe}} u$ and ES *left for* $t \to_{\mathsf{oe}} p$, *i.e.* $t \coloneqq q[x \leftarrow L\langle v\rangle] \mapsto_{\mathsf{e}} L\langle q\{x \leftarrow v\}\rangle =: u$ and $t \mapsto_{\mathsf{e}} q'[x \leftarrow L\langle v\rangle] =: p$ with $q \to_{\mathsf{oe}} q'$: then,
$$u \to_{\mathsf{oe}} L\langle q'[x \leftarrow v]\rangle \ {}_{\mathsf{oe}}\!\leftarrow p$$
- *Step at the Root for* $t \to_{\mathsf{oe}} u$ and ES *right for* $t \to_{\mathsf{oe}} p$, *i.e.*, for some $n > 0$, $t \coloneqq q[x \leftarrow v[x_1 \leftarrow t_1] \dots [x_n \leftarrow t_n]] \mapsto_{\mathsf{e}} q\{x \leftarrow v\}[x_1 \leftarrow t_1] \dots [x_n \leftarrow t_n] =: u$ whereas $t \to_{\mathsf{oe}} q[x \leftarrow v[x_1 \leftarrow t_1] \dots [x_j \leftarrow t'_j] \dots [x_n \leftarrow t_n]] =: p$ with $t_j \to_{\mathsf{oe}} t'_j$ for some $1 \leq j \leq n$: then,
$$u \to_{\mathsf{oe}} q\{x \leftarrow v\}[x_1 \leftarrow t_1] \dots [x_j \leftarrow t'_j] \dots [x_n \leftarrow t_n] \ {}_{\mathsf{oe}}\!\leftarrow p;$$
- *Application Left for* $t \to_{\mathsf{oe}} u$ and *Application Right for* $t \to_{\mathsf{oe}} p$, *i.e.* $t \coloneqq qr \to_{\mathsf{oe}} q'r =: u$ and $t \to_{\mathsf{oe}} qr' =: p$ with $q \to_{\mathsf{oe}} q'$ and $r \to_{\mathsf{oe}} r'$: then, $u \to_{\mathsf{oe}} q'r' \ {}_{\mathsf{oe}}\!\leftarrow p$;
- *Application Left for both* $t \to_{\mathsf{oe}} u$ *and* $t \to_{\mathsf{oe}} p$, *i.e.* $t \coloneqq qr \to_{\mathsf{oe}} q'r =: u$ and $t \to_{\mathsf{oe}} q''r =: p$ with $q' \ {}_{\mathsf{oe}}\!\leftarrow q \to_{\mathsf{oe}} q''$: by *i.h.*, there exists $q_0 \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{oe}} q_0 \ {}_{\mathsf{oe}}\!\leftarrow q''$, hence $u \to_{\mathsf{oe}} q_0 r \ {}_{\mathsf{oe}}\!\leftarrow p$;
- *Application Right for both* $t \to_{\mathsf{oe}} u$ *and* $t \to_{\mathsf{oe}} p$, *i.e.* $t \coloneqq rq \to_{\mathsf{oe}} rq' =: u$ and $t \to_{\mathsf{oe}} rq'' =: p$ with $q' \ {}_{\mathsf{oe}}\!\leftarrow q \to_{\mathsf{oe}} q''$: by *i.h.*, there exists $q_0 \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{oe}} q_0 \ {}_{\mathsf{oe}}\!\leftarrow q''$, hence $u \to_{\mathsf{oe}} rq_0 \ {}_{\mathsf{oe}}\!\leftarrow p$;
- ES *left for* $t \to_{\mathsf{oe}} u$ and ES *right for* $t \to_{\mathsf{oe}} p$, *i.e.* $t \coloneqq q[x \leftarrow r] \to_{\mathsf{oe}} q'[x \leftarrow r] =: u$ and $t \to_{\mathsf{oe}} q[x \leftarrow r'] =: p$ with $q \to_{\mathsf{oe}} q'$ and $r \to_{\mathsf{oe}} r'$: then, $u \to_{\mathsf{oe}} q'[x \leftarrow r'] \ {}_{\mathsf{oe}}\!\leftarrow p$;
- ES *left for both* $t \to_{\mathsf{e}} u$ *and* $t \to_{\mathsf{e}} p$, *i.e.* $t \coloneqq q[x \leftarrow r] \to_{\mathsf{e}} q'[x \leftarrow r] =: u$ and $t \to_{\mathsf{e}} q''[x \leftarrow r] =: p$ with $q' \ {}_{\mathsf{e}}\!\leftarrow q \to_{\mathsf{e}} q''$: by *i.h.*, there exists $q_0 \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{e}} q_0 \ {}_{\mathsf{e}}\!\leftarrow q''$, hence $u \to_{\mathsf{e}} q_0[x \leftarrow r] \ {}_{\mathsf{e}}\!\leftarrow p$;
- ES *right for both* $t \to_{\mathsf{e}} u$ *and* $t \to_{\mathsf{e}} p$, *i.e.* $t \coloneqq r[x \leftarrow q] \to_{\mathsf{e}} r[x \leftarrow q'] =: u$ and $t \to_{\mathsf{e}} r[x \leftarrow q''] =: p$ with $q' \ {}_{\mathsf{e}}\!\leftarrow q \to_{\mathsf{e}} q''$: by *i.h.*, there exists $q_0 \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{e}} q_0 \ {}_{\mathsf{e}}\!\leftarrow q''$, hence $u \to_{\mathsf{e}} r[x \leftarrow q_0] \ {}_{\mathsf{e}}\!\leftarrow p$.

Note that in [11, Lemma 11] it has just been proved the strong confluence of $\to_{\mathsf{vsc}}$, not of $\to_{\mathsf{m}}$ or $\to_{\mathsf{e}}$.

3) We show that $\to_{\mathsf{oe}}$ and $\to_{\mathsf{om}}$ strongly commute, *i.e.* if $u \ {}_{\mathsf{oe}}\!\leftarrow t \to_{\mathsf{om}} p$, then $u \neq p$ and there is $t' \in \Lambda_{\mathsf{vsc}}$ such that $u \to_{\mathsf{om}} t' \ {}_{\mathsf{oe}}\!\leftarrow p$. The proof is by induction on the definition of $t \to_{\mathsf{oe}} u$. The proof that $u \neq p$ is left to the reader. Since the $\to_{\mathsf{e}}$ and $\to_{\mathsf{m}}$ cannot reduce under λ's, all values are om-normal and oe-normal. So, there are the following cases.

- *Step at the Root for* $t \to_{\mathsf{oe}} u$ and ES *left for* $t \to_{\mathsf{om}} p$, *i.e.* $t \coloneqq q[z \leftarrow L\langle v\rangle] \to_{\mathsf{oe}} L\langle q\{z \leftarrow v\}\rangle =: u$ and $t \to_{\mathsf{om}} q'[z \leftarrow L\langle v\rangle] =: p$ with $q \to_{\mathsf{om}} q'$: then
$$u \to_{\mathsf{om}} L\langle q'\{z \leftarrow v\}\rangle \ {}_{\mathsf{oe}}\!\leftarrow u$$
- *Step at the Root for* $t \to_{\mathsf{oe}} u$ and ES *right for* $t \to_{\mathsf{om}} p$, *i.e.*
$$\begin{aligned} t &\coloneqq q[z \leftarrow v[x_1 \leftarrow t_1] \dots [x_n \leftarrow t_n]] \\ &\to_{\mathsf{oe}} q\{z \leftarrow v\}[x_1 \leftarrow t_1] \dots [x_n \leftarrow t_n] =: u \end{aligned}$$
and $t \to_{\mathsf{om}} q[z \leftarrow v[x_1 \leftarrow t_1] \dots [x_j \leftarrow t'_j] \dots [x_n \leftarrow t_n]] =: p$ for some $n > 0$, and $t_j \to_{\mathsf{om}} t'_j$ for some $1 \leq j \leq n$: then, $u \to_{\mathsf{om}} q\{z \leftarrow v\}[x_1 \leftarrow t_1] \dots [x_j \leftarrow t'_j] \dots [x_n \leftarrow t_n] \ {}_{\mathsf{oe}}\!\leftarrow p$;
- *Application Left for* $t \to_{\mathsf{e}} u$ and *Application Right for* $t \to_{\mathsf{m}} p$, *i.e.* $t \coloneqq qr \to_{\mathsf{oe}} q'r =: u$ and $t \to_{\mathsf{om}} qr' =: p$ with $q \to_{\mathsf{e}} q'$ and $r \to_{\mathsf{om}} r'$: then, $t \to_{\mathsf{om}} q'r' \ {}_{\mathsf{oe}}\!\leftarrow u$;
- *Application Left for both* $t \to_{\mathsf{oe}} u$ *and* $t \to_{\mathsf{om}} p$, *i.e.* $t \coloneqq qr \to_{\mathsf{oe}} q'r =: u$ and $t \to_{\mathsf{om}} q''r =: p$ with $q' \ {}_{\mathsf{oe}}\!\leftarrow q \to_{\mathsf{om}} q''$: by *i.h.*, there exists $s \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{om}} s \ {}_{\mathsf{oe}}\!\leftarrow q''$, hence $u \to_{\mathsf{om}} sr \ {}_{\mathsf{oe}}\!\leftarrow p$;
- *Application Left for* $t \to_{\mathsf{oe}} u$ and *Step at the Root for* $t \to_{\mathsf{om}} p$, *i.e.* $t \coloneqq (\lambda x.r)[x_1 \leftarrow t_1] \dots [x_n \leftarrow t_n]q \to_{\mathsf{oe}} (\lambda x.r)[x_1 \leftarrow t_1] \dots [x_j \leftarrow t'_j] \dots [x_n \leftarrow t_n]q =: u$ with $n > 0$ and $t_j \to_{\mathsf{oe}} t'_j$ for some $1 \leq j \leq n$, and
$$t \to_{\mathsf{om}} r[x \leftarrow q][x_1 \leftarrow t_1] \dots [x_n \leftarrow t_n] =: p$$
Then,
$$u \to_{\mathsf{om}} r[x \leftarrow q][x_1 \leftarrow t_1] \dots [x_j \leftarrow t'_j] \dots [x_n \leftarrow t_n] \ {}_{\mathsf{oe}}\!\leftarrow p;$$
- *Application Right for* $t \to_{\mathsf{oe}} u$ and *Application Left for* $t \to_{\mathsf{m}} p$, *i.e.* $t \coloneqq rq \to_{\mathsf{oe}} rq' =: u$ and $t \to_{\mathsf{om}} r'q =: p$ with $q \to_{\mathsf{oe}} q'$ and $r \to_{\mathsf{om}} r'$: then, $u \to_{\mathsf{om}} r'q' \ {}_{\mathsf{oe}}\!\leftarrow p$;
- *Application Right for both* $t \to_{\mathsf{oe}} u$ *and* $t \to_{\mathsf{om}} p$, *i.e.* $t \coloneqq rq \to_{\mathsf{oe}} rq' =: u$ and $t \to_{\mathsf{om}} rq'' =: p$ with $q' \ {}_{\mathsf{oe}}\!\leftarrow q \to_{\mathsf{om}} q''$: by *i.h.*, there exists $s \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{om}} s \ {}_{\mathsf{oe}}\!\leftarrow q''$, hence $u \to_{\mathsf{om}} rs \ {}_{\mathsf{oe}}\!\leftarrow p$;
- *Application Right for* $t \to_{\mathsf{oe}} u$ and *Step at the Root for* $t \to_{\mathsf{om}} p$, *i.e.* $t \coloneqq L\langle \lambda x.r\rangle q \to_{\mathsf{oe}} L\langle \lambda x.r\rangle q' =: u$ with $q \to_{\mathsf{oe}} q'$, and $t \to_{\mathsf{om}} L\langle r[x \leftarrow q]\rangle =: p$: then, $u \to_{\mathsf{om}} L\langle r[x \leftarrow q']\rangle \ {}_{\mathsf{oe}}\!\leftarrow p$;

- ES *left for* $t \to_{\mathsf{oe}} u$ and ES *right for* $t \to_{\mathsf{om}} p$, *i.e.* $t \coloneqq q[x\leftarrow r] \to_{\mathsf{oe}} q'[x\leftarrow r] =: u$ and $t \to_{\mathsf{om}} q[x\leftarrow r'] =: p$ with $q \to_{\mathsf{e}} q'$ and $r \to_{\mathsf{om}} r'$: then, $u \to_{\mathsf{om}} q'[x\leftarrow r'] \,_{\mathsf{oe}}\!\leftarrow p$;
- ES *left for both* $t \to_{\mathsf{oe}} u$ *and* $t \to_{\mathsf{om}} p$, *i.e.* $t \coloneqq q[x\leftarrow r] \to_{\mathsf{e}} q'[x\leftarrow r] =: u$ and $t \to_{\mathsf{om}} q''[x\leftarrow r] =: p$ with $q' \,_{\mathsf{oe}}\!\leftarrow q \to_{\mathsf{om}} q''$: by *i.h.*,there is $s \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{om}} s \,_{\mathsf{oe}}\!\leftarrow q''$, hence $u \to_{\mathsf{om}} s[x\leftarrow r] \,_{\mathsf{oe}}\!\leftarrow p$;
- ES *right for* $t \to_{\mathsf{oe}} u$ and ES *left for* $t \to_{\mathsf{om}} p$, *i.e.* $t \coloneqq r[x\leftarrow q] \to_{\mathsf{oe}} r[x\leftarrow q'] =: u$ and $t \to_{\mathsf{om}} r'[x\leftarrow q] =: p$ with $q \to_{\mathsf{oe}} q'$ and $r \to_{\mathsf{om}} r'$: then, $u \to_{\mathsf{om}} r'[x\leftarrow q'] \,_{\mathsf{oe}}\!\leftarrow p$;
- ES *right for both* $t \to_{\mathsf{oe}} u$ *and* $t \to_{\mathsf{om}} p$, *i.e.* $t \coloneqq r[x\leftarrow q] \to_{\mathsf{oe}} r[x\leftarrow q'] =: u$ and $t \to_{\mathsf{om}} r[x\leftarrow q''] =: p$ with $q \,_{\mathsf{e}}\!\leftarrow q' \to_{\mathsf{om}} q''$: by *i.h.*, there is $s \in \Lambda_{\mathsf{vsc}}$ such that $q \to_{\mathsf{om}} s \,_{\mathsf{oe}}\!\leftarrow q''$, so $u \to_{\mathsf{om}} r[x\leftarrow s] \,_{\mathsf{oe}}\!\leftarrow p$. □

APPENDIX C
PROOFS OF SECTION III (EXTERNAL STRATEGY)

Lemma C.1 (Properties of rigid terms)**.**

1) Given $t \in \Lambda_{\mathsf{vsc}}$ and a rigid context R, $R\langle t\rangle$ is a rigid term.
2) Let r be a rigid term and $t \in \Lambda_{\mathsf{vsc}}$ such that $r \to_{\mathsf{o}} t$. Then t is rigid.
3) Let r be a rigid term and $t \in \Lambda_{\mathsf{vsc}}$ such that $r \to_{\mathsf{x}} t$. Then t is rigid.

Proof.

1) By induction on the definition of R.
2) Let O be an open evaluation context such that $r = O\langle u\rangle \to_{\mathsf{o}} O\langle u'\rangle = t$, with $u \mapsto_{\mathsf{m}} u'$ or $u \mapsto_{\mathsf{e}} u'$. We prove this for $\to_{\mathsf{om}}$ by structural induction on O; the proof for $\to_{\mathsf{oe}}$ follows the same schema.
   - *Empty context*; *i.e.*, $O = \langle\cdot\rangle$. This case is not possible, because it would imply that $r = u = L\langle\lambda x.p\rangle q$, which is not a rigid term.
   - *Application right*; *i.e.*, $O = qO'$. As $r = qO'\langle u\rangle$, then q is a rigid term, and so $qO'\langle u'\rangle = t$ is rigid.
   - *Application left*; *i.e.*, $O = O'q$. As $r = O'\langle u\rangle q$ and $O'\langle u'\rangle$ is rigid by *i.h.*, then $O'\langle u'\rangle q = t$ is rigid too.
   - ES *left*; *i.e.*, $O = O'[x\leftarrow q]$. Since $r = O'\langle u\rangle[x\leftarrow q]$, then both $O'\langle u\rangle$ and q are rigid. Moreover, $O'\langle u'\rangle$ is rigid by *i.h.*, and so $O'\langle u'\rangle[x\leftarrow q] = t$ is rigid too.
   - ES *right*; *i.e.*, $O = q[x\leftarrow O']$. Since $r = q[x\leftarrow O'\langle u\rangle]$, then both q and $O'\langle u\rangle$ are rigid. Moreover, $O'\langle u'\rangle$ is rigid by *i.h.*, and so $q[x\leftarrow O'\langle u'\rangle] = t$ is rigid too.
3) Let S be a strong evaluation context such that $r = S\langle u\rangle \to_{\mathsf{x}} S\langle u'\rangle = t$, with $u \to_{\mathsf{om}} u'$ or $u \to_{\mathsf{oe}} u'$. We prove this for $u \to_{\mathsf{om}} u'$ by structural induction on S; the proof for $\to_{\mathsf{oe}}$ follows the same schema.
   - *Empty context*; *i.e.*, $S = \langle\cdot\rangle$. Then $r = u \to_{\mathsf{om}} u' = t$ and the statement holds by Lemma C.1.2
   - *Under λ-abstraction right*; *i.e.*, $S = \lambda x.S'$. This case is not possible, because it would imply that $r = \lambda x.S'\langle u\rangle$, which is not a rigid term.
   - *External context,* ES *right*; *i.e.*, $S = q[x\leftarrow R]$. Since $r = q[x\leftarrow R\langle u\rangle]$, then both q and $R\langle u\rangle$ are rigid terms. Moreover, $R\langle u'\rangle$ is rigid by *i.h.*, and so $q[x\leftarrow R\langle u'\rangle] = p$ is rigid too.
   - *External context,* ES *left*; *i.e.*, $S = S'[x\leftarrow q]$, with q a rigid term. Since $r = S'\langle u\rangle[x\leftarrow q]$, then $S'\langle u\rangle$ is rigid. Moreover, $S'\langle u'\rangle$ is rigid by *i.h.*, and so $S'\langle u'\rangle[x\leftarrow q] = p$ is rigid too.
   - *Rigid context, application right*; *i.e.*, $S = qS'$, with q a rigid term. Then $qS'\langle u'\rangle = p$ is rigid too.
   - *Rigid context, application left*; *i.e.*, $S = Rq$. Since $r = R\langle u\rangle q$, then $R\langle u\rangle$ is rigid. Moreover, $R\langle u'\rangle$ is rigid by *i.h.*, and so $R\langle u'\rangle q$ is rigid too.
   - *Rigid context,* ES *left*; *i.e.*, $S = R[x\leftarrow q]$, with q a rigid term. Since $r = R\langle u\rangle[x\leftarrow q]$, then $R\langle u\rangle$ is rigid. Moreover, $R\langle u'\rangle$ is rigid by *i.h.*, and so $R\langle u'\rangle[x\leftarrow q] = p$ is rigid too.
   - *Rigid context,* ES *right*; *i.e.*, $S = q[x\leftarrow R]$, with q a rigid term. Since $r = q[x\leftarrow R\langle u\rangle]$, then $R\langle u\rangle$ is rigid. Moreover, $R\langle u'\rangle$ is rigid by *i.h.*, and so $q[x\leftarrow R\langle u'\rangle] = p$ is rigid too.

□

The properties stated by the next proposition are the building blocks for the proof that $\to_{\mathsf{x}}$ is diamond, coming right next. In particular, the strong commutation of $\to_{\mathsf{xm}}$ and $\to_{\mathsf{xe}}$ shall ensure that the diamond of $\to_{\mathsf{x}}$ preserves the kind of step, thus strengthening the diamond of $\to_{\mathsf{x}}$ with the guarantee that all $\to_{\mathsf{x}}$ sequences to normal form (if any) have the same number of $\to_{\mathsf{m}}$ steps.

Lemma C.2 (Basic Properties of $\to_{\mathsf{x}}$)**.**

1) $\to_{\mathsf{xm}}$ and $\to_{\mathsf{xe}}$ are diamond (separately).
2) $\to_{\mathsf{xm}}$ and $\to_{\mathsf{xe}}$ strongly commute.

Proof. 1) We prove that $\to_{\mathsf{xm}}$ is diamond, *i.e.* if $u \,_{\mathsf{xm}}\!\leftarrow t \to_{\mathsf{xm}} p$ with $u \neq p$ then there exists $t' \in \Lambda_{\mathsf{vsc}}$ such that $u \to_{\mathsf{xm}} t' \,_{\mathsf{xm}}\!\leftarrow p$ (the proof that $\to_{\mathsf{xe}}$ is diamond is analogue). The proof is by structural induction on t, doing case analysis on $t \to_{\mathsf{xm}} u$ and $t \to_{\mathsf{xm}} p$:

- *Under λ-abstraction for both $t \to_{\mathsf{xm}} u$ and $t \to_{\mathsf{xm}} p$; i.e.,* $t = \lambda x.q \to_{\mathsf{xm}} \lambda x.r = u$ and $t = \lambda x.q \to_{\mathsf{xm}} \lambda x.s = p$, with $q \to_{\mathsf{xm}} r$ and $q \to_{\mathsf{xm}} s$. By *i.h.* there exists q' such that $r \to_{\mathsf{xm}} q' \ {}_{\mathsf{xm}}\!\leftarrow s$ and so $u = \lambda x.r \to_{\mathsf{xm}} \lambda x.q' \ {}_{\mathsf{xm}}\!\leftarrow \lambda x.s = p$.
- *Application right for $t \to_{\mathsf{xm}} u$ and application left for $t \to_{\mathsf{xm}} p$; i.e.,* $t = qr \to_{\mathsf{xm}} qr' = u$ and $t = qr \to_{\mathsf{xm}} q'r = p$. There are several sub-cases to this:
  - Let $t = qr = qO\langle\tilde{r}\rangle \to_{\mathsf{om}} qO\langle\tilde{r'}\rangle = u$, with $\tilde{r} \mapsto_{\mathsf{m}} \tilde{r'}$, and $t = R\langle\tilde{q}\rangle r \to_{\mathsf{xm}} R\langle\tilde{q'}\rangle r$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$. Let $t' \coloneqq R\langle\tilde{q'}\rangle O\langle\tilde{r'}\rangle$, having that
  $$u = R\langle\tilde{q}\rangle O\langle\tilde{r'}\rangle \to_{\mathsf{xm}} t' \ {}_{\mathsf{xm}}\!\leftarrow R\langle\tilde{q'}\rangle O\langle\tilde{r}\rangle = p$$
  Note that $u \to_{\mathsf{xm}} t'$ holds because every rigid context is an open context.
  - Let $t = qr = qO_1\langle\tilde{r}\rangle \to_{\mathsf{om}} qO_1\langle\tilde{r'}\rangle = u$, with $\tilde{r} \mapsto_{\mathsf{m}} \tilde{r'}$, and $t = O_2\langle\tilde{q}\rangle r \to_{\mathsf{om}} O_2\langle\tilde{q'}\rangle r$, with $\tilde{q} \mapsto_{\mathsf{m}} \tilde{q'}$. Then the statement holds by Lemma B.3.2
  - Let $t = qS\langle\tilde{r}\rangle \to_{\mathsf{xm}} qS\langle\tilde{r'}\rangle = u$, with q a rigid term and $\tilde{r} \to_{\mathsf{om}} \tilde{r'}$, and $t = R\langle\tilde{q}\rangle r \to_{\mathsf{xm}} R\langle\tilde{q'}\rangle r = p$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$. Let $t' \coloneqq R\langle\tilde{q'}\rangle S\langle\tilde{r'}\rangle$, having that
  $$u = R\langle\tilde{q}\rangle S\langle\tilde{r'}\rangle \to_{\mathsf{xm}} t' \ {}_{\mathsf{xm}}\!\leftarrow R\langle\tilde{q'}\rangle S\langle\tilde{r}\rangle = p$$
  Note that $t' \ {}_{\mathsf{xm}}\!\leftarrow p$ holds because $R\langle\tilde{q'}\rangle$ is a rigid term —by Lemma C.1.1.
  - Let $t = qS\langle\tilde{r}\rangle \to_{\mathsf{xm}} qS\langle\tilde{r'}\rangle = u$, with q a rigid term and $\tilde{r} \to_{\mathsf{om}} \tilde{r'}$, and $t = O\langle\tilde{q}\rangle r \to_{\mathsf{xm}} O\langle\tilde{q'}\rangle r$, with $\tilde{q} \mapsto_{\mathsf{m}} \tilde{q'}$. Let $t' \coloneqq O\langle\tilde{q'}\rangle S\langle\tilde{r'}\rangle$, having that
  $$u = O\langle\tilde{q}\rangle S\langle\tilde{r'}\rangle \to_{\mathsf{xm}} t' \ {}_{\mathsf{xm}}\!\leftarrow O\langle\tilde{q'}\rangle S\langle\tilde{r}\rangle = p$$
  Note that $t' \ {}_{\mathsf{xm}}\!\leftarrow p$ holds because the fact that q is a rigid term and that $q = O\langle\tilde{q}\rangle \to_{\mathsf{om}} O\langle\tilde{q'}\rangle$ imply that $O\langle\tilde{q'}\rangle$ is a rigid term —by Lemma C.1.2.
- *Application right for both $t \to_{\mathsf{xm}} u$ and $t \to_{\mathsf{xm}} p$; i.e.,* $t = qr \to_{\mathsf{xm}} qr' = u$ and $t = qr \to_{\mathsf{xm}} qr'' = p$. By *i.h.* there exists $s \in \Lambda_{\mathsf{vsc}}$ such that $r' \to_{\mathsf{xm}} s \ {}_{\mathsf{xm}}\!\leftarrow r''$. The analysis of the sub-cases, depending on the open/strong/rigid type contexts involved in $t \to_{\mathsf{xm}} u$ and $t \to_{\mathsf{xm}} p$, follows the same schema as for the previous item, all showing that
  $$u = qr' \to_{\mathsf{xm}} qs \ {}_{\mathsf{xm}}\!\leftarrow qr'' = p$$
- *Application left for both $t \to_{\mathsf{xm}} u$ and $t \to_{\mathsf{xm}} p$; i.e.,* $t = qr \to_{\mathsf{xm}} q'r = u$ and $t = qr \to_{\mathsf{xm}} q''r = p$. By *i.h.* there exists $s \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{xm}} s \ {}_{\mathsf{xm}}\!\leftarrow q''$. The analysis of the sub-cases, depending on the open/strong/rigid type contexts involved in $t \to_{\mathsf{xm}} u$ and $t \to_{\mathsf{xm}} p$, follows the same schema as for the previous item, all showing that
  $$u = q'r \to_{\mathsf{xm}} sr \ {}_{\mathsf{xm}}\!\leftarrow q''r = p$$
- ES *right for $t \to_{\mathsf{xm}} u$ and* ES *left for $t \to_{\mathsf{xm}} p$; i.e.,* $t = q[x{\leftarrow}r] \to_{\mathsf{xm}} q[x{\leftarrow}r'] = u$ and $t = q[x{\leftarrow}r] \to_{\mathsf{xm}} q'[x{\leftarrow}r]$. There are several sub-cases to this:
  - Let $t = q[x{\leftarrow}O\langle\tilde{r}\rangle] \to_{\mathsf{xm}} q[x{\leftarrow}O\langle\tilde{r'}\rangle] = u$, with $\tilde{r} \mapsto_{\mathsf{m}} \tilde{r'}$, and $t = O\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} O\langle\tilde{q'}\rangle[x{\leftarrow}r] = p$, with $\tilde{q} \mapsto_{\mathsf{m}} \tilde{q'}$. Then the statement holds by Lemma B.3.2.
  - Let $t = q[x{\leftarrow}O\langle\tilde{r}\rangle] \to_{\mathsf{xm}} q[x{\leftarrow}O\langle\tilde{r'}\rangle] = u$, with $\tilde{r} \mapsto_{\mathsf{m}} \tilde{r'}$, and $t = S\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} S\langle\tilde{q'}\rangle[x{\leftarrow}r] = p$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let $t' \coloneqq S\langle\tilde{q'}\rangle[x{\leftarrow}O\langle\tilde{r'}\rangle]$, having that
  $$u = S\langle\tilde{q}\rangle[x{\leftarrow}O\langle\tilde{r'}\rangle] \to_{\mathsf{xm}} t' \ {}_{\mathsf{xm}}\!\leftarrow S\langle\tilde{q'}\rangle[x{\leftarrow}O\langle\tilde{r}\rangle] = p$$
  Note that $u \to_{\mathsf{xm}} t'$ holds because the fact that$r$ is a rigid term and that $r = O\langle\tilde{r}\rangle \to_{\mathsf{xm}} O\langle\tilde{r'}\rangle$ imply that $O\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.3.
  - Let $t = q[x{\leftarrow}O\langle\tilde{r}\rangle] \to_{\mathsf{xm}} q[x{\leftarrow}O\langle\tilde{r'}\rangle] = u$, with $\tilde{r} \mapsto_{\mathsf{m}} \tilde{r'}$, and $t = R\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} R\langle\tilde{q'}\rangle[x{\leftarrow}r] = p$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let $t' \coloneqq R\langle\tilde{q'}\rangle[x{\leftarrow}O\langle\tilde{r'}\rangle]$, having that
  $$u = R\langle\tilde{q}\rangle[x{\leftarrow}O\langle\tilde{r'}\rangle] \to_{\mathsf{xm}} t' \ {}_{\mathsf{xm}}\!\leftarrow R\langle\tilde{q'}\rangle[x{\leftarrow}O\langle\tilde{r}\rangle] = p$$
  Note that $u \to_{\mathsf{xm}} t'$ holds because the fact that r is a rigid term and that $r = O\langle\tilde{r}\rangle \to_{\mathsf{xm}} O\langle\tilde{r'}\rangle$ imply $O\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.2.
  - Let $t = q[x{\leftarrow}R\langle\tilde{r}\rangle] \to_{\mathsf{xm}} q[x{\leftarrow}R\langle\tilde{r'}\rangle] = u$, with $\tilde{r} \to_{\mathsf{om}} \tilde{r'}$, and $t = O\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} O\langle\tilde{q'}\rangle[x{\leftarrow}r] = p$, with $\tilde{q} \mapsto_{\mathsf{m}} \tilde{q'}$. Let $t' \coloneqq O\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle]$, having that
  $$u = O\langle\tilde{q}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle] \to_{\mathsf{xm}} t' \ {}_{\mathsf{xm}}\!\leftarrow O\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r}\rangle] = p$$
  - Let $t = q[x{\leftarrow}R\langle\tilde{r}\rangle] \to_{\mathsf{xm}} q[x{\leftarrow}R\langle\tilde{r'}\rangle] = u$, with $\tilde{r} \to_{\mathsf{om}} \tilde{r'}$, and $t = S\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} S\langle\tilde{q'}\rangle[x{\leftarrow}r] = p$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let $t' \coloneqq S\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle]$, having that
  $$u = S\langle\tilde{q}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle] \to_{\mathsf{xm}} t' \ {}_{\mathsf{xm}}\!\leftarrow S\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r}\rangle] = p$$

Note that $u \to_{\mathsf{xm}} t'$ holds because the fact that r is a rigid term and that $r = R\langle\tilde{r}\rangle \to_{\mathsf{xm}} R\langle\tilde{r'}\rangle$ imply that $R\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.3.

– Let $t = q[x{\leftarrow}R_1\langle\tilde{r}\rangle] \to_{\mathsf{xm}} q[x{\leftarrow}R_1\langle\tilde{r'}\rangle] = u$, with $\tilde{r} \to_{\mathsf{om}} \tilde{r'}$, and $t = R_2\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} R_2\langle\tilde{q'}\rangle[x{\leftarrow}r] = p$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let $t' \coloneqq R_2\langle\tilde{q'}\rangle[x{\leftarrow}R_1\langle\tilde{r'}\rangle]$, having that

$$u = R_2\langle\tilde{q}\rangle[x{\leftarrow}R_1\langle\tilde{r'}\rangle] \to_{\mathsf{xm}} t' \;{}_{\mathsf{xm}}{\leftarrow}\; R_2\langle\tilde{q'}\rangle[x{\leftarrow}R_1\langle\tilde{r}\rangle] = p$$

Note that $u \to_{\mathsf{xm}} t'$ holds because the fact that r is a rigid term and that $r = R_1\langle\tilde{r}\rangle \to_{\mathsf{xm}} R_1\langle\tilde{r'}\rangle$ imply that $R_1\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.3.

– Let $t = q[x{\leftarrow}R\langle\tilde{r}\rangle] \to_{\mathsf{xm}} q[x{\leftarrow}R\langle\tilde{r'}\rangle] = u$, with $\tilde{r} \to_{\mathsf{om}} \tilde{r'}$ and q is a rigid term, and $t = O\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} O\langle\tilde{q'}\rangle[x{\leftarrow}r] = p$, with $\tilde{q} \mapsto_{\mathsf{m}} \tilde{q'}$. Let $t' \coloneqq O\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle]$, having that

$$u = O\langle\tilde{q}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle] \to_{\mathsf{xm}} t' \;{}_{\mathsf{xm}}{\leftarrow}\; O\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r}\rangle] = p$$

Note that $p \;{}_{\mathsf{xm}}{\leftarrow}\; t'$ holds because the fact that r is a rigid term and that $r = R\langle\tilde{r}\rangle \to_{\mathsf{xm}} R\langle\tilde{r'}\rangle$ imply that $R\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.3.

– Let $t = q[x{\leftarrow}R\langle\tilde{r}\rangle] \to_{\mathsf{xm}} q[x{\leftarrow}R\langle\tilde{r'}\rangle] = u$, with $\tilde{r} \to_{\mathsf{om}} \tilde{r'}$ and q is a rigid term, and $t = S\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} S\langle\tilde{q'}\rangle[x{\leftarrow}r] = p$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let $t' \coloneqq S\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle]$, having that

$$u = S\langle\tilde{q}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle] \to_{\mathsf{xm}} t' \;{}_{\mathsf{xm}}{\leftarrow}\; S\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r}\rangle] = p$$

Note that $u \to_{\mathsf{xm}} t'$ holds because the fact that r is a rigid term and that $r = R\langle\tilde{r}\rangle \to_{\mathsf{xm}} R\langle\tilde{r'}\rangle$ imply that $R\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.3. Moreover, note that $t' \;{}_{\mathsf{xm}}{\leftarrow}\; p$ holds because the fact that q is a rigid term and that $q = S\langle\tilde{q}\rangle \to_{\mathsf{xm}} S\langle\tilde{q'}\rangle$ imply that $S\langle\tilde{q'}\rangle$ is a rigid term —by Lemma C.1.3.

– Let $t = q[x{\leftarrow}R\langle\tilde{r}\rangle] \to_{\mathsf{xm}} q[x{\leftarrow}R\langle\tilde{r'}\rangle] = u$, with $\tilde{r} \to_{\mathsf{om}} \tilde{r'}$ and q is a rigid term, and $t = R\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} R\langle\tilde{q'}\rangle[x{\leftarrow}r] = p$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let $t' \coloneqq R\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle]$, having that

$$u = R\langle\tilde{q}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle] \to_{\mathsf{xm}} t' \;{}_{\mathsf{xm}}{\leftarrow}\; R\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r}\rangle] = p$$

Note that $u \to_{\mathsf{xm}} t'$ holds because the fact that r is a rigid term and that $r = R\langle\tilde{r}\rangle \to_{\mathsf{xm}} R\langle\tilde{r'}\rangle$ imply that $R\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.3. Moreover, note that $t' \;{}_{\mathsf{xm}}{\leftarrow}\; p$ holds because the fact that q is a rigid term and that $q = R\langle\tilde{q}\rangle \to_{\mathsf{xm}} R\langle\tilde{q'}\rangle$ imply that $R\langle\tilde{q'}\rangle$ is a rigid term —by Lemma C.1.3.

- ES *right for both* $t \to_{\mathsf{xm}} u$ *and* $t \to_{\mathsf{xm}} p$; *i.e.*, $t = q[x{\leftarrow}r] \to_{\mathsf{xm}} q[x{\leftarrow}r'] = u$ and $t = q[x{\leftarrow}r] \to_{\mathsf{xm}} q[x{\leftarrow}r''] = p$. By *i.h.* there exists $s \in \Lambda_{\mathsf{vsc}}$ such that $r' \to_{\mathsf{xm}} s \;{}_{\mathsf{xm}}{\leftarrow}\; r''$. The analysis of the sub-cases, depending on the open/strong/rigid type contexts involved in $t \to_{\mathsf{xm}} u$ and $t \to_{\mathsf{xm}} p$, follows the same schema as for the previous item, all showing that

$$u = q[x{\leftarrow}r'] \to_{\mathsf{xm}} q[x{\leftarrow}s] \;{}_{\mathsf{xm}}{\leftarrow}\; q[x{\leftarrow}r''] = p$$

- ES *left for both* $t \to_{\mathsf{xm}} u$ *and* $t \to_{\mathsf{xm}} p$; *i.e.*, $t = q[x{\leftarrow}r] \to_{\mathsf{xm}} q'[x{\leftarrow}r] = u$ and $t = q''[x{\leftarrow}r]$. By *i.h.* there exists $s \in \Lambda_{\mathsf{vsc}}$ such that $q' \to_{\mathsf{xm}} s \;{}_{\mathsf{xm}}{\leftarrow}\; q''$. The analysis of the sub-cases, depending on the open/strong/rigid type contexts involved in $t \to_{\mathsf{xm}} u$ and $t \to_{\mathsf{xm}} p$, follows the same schema as for the previous item, all showing that

$$u = q'[x{\leftarrow}r] \to_{\mathsf{xm}} s[x{\leftarrow}r] \;{}_{\mathsf{xm}}{\leftarrow}\; q''[x{\leftarrow}r] = p$$

The proof that $\to_{\mathsf{xe}}$ is diamond (*i.e.*, if $u \;{}_{\mathsf{xe}}{\leftarrow}\; t \to_{\mathsf{xe}} p$ with $u \neq p$ then there exists $t' \in \Lambda_{\mathsf{vsc}}$ such that $u \to_{\mathsf{xe}} t' \;{}_{\mathsf{xe}}{\leftarrow}\; p$) follows the same schema as for $\to_{\mathsf{xm}}$.

2) We show that $\to_{\mathsf{xe}}$ and $\to_{\mathsf{xm}}$ strongly commute; *i.e.*, if $u \;{}_{\mathsf{xe}}{\leftarrow}\; t \to_{\mathsf{xm}} p$, then $u \neq p$ and there is $t' \in \Lambda_{\mathsf{vsc}}$ such that $u \to_{\mathsf{xm}} t' \;{}_{\mathsf{xe}}{\leftarrow}\; p$. The proof is by structural induction on t, doing case analysis on $t \;{}_{\mathsf{xe}}{\leftarrow}\; u$ and $t \to_{\mathsf{xm}} p$:

- *Under* λ*-abstraction for both* $t \;{}_{\mathsf{xe}}{\leftarrow}\; u$ *and* $t \to_{\mathsf{xm}} p$; *i.e.*, $t = \lambda x.r \;{}_{\mathsf{xe}}{\leftarrow}\; \lambda x.q = u$ and $t = \lambda x.q \to_{\mathsf{xm}} \lambda x.s = p$, with $r \;{}_{\mathsf{oe}}{\leftarrow}\; q \to_{\mathsf{om}} s$. By Lemma B.3.3, there exists q' such that $r \to_{\mathsf{xm}} q' \;{}_{\mathsf{xe}}{\leftarrow}\; s$, and so $u = \lambda x.r \to_{\mathsf{om}} \lambda x.q' \;{}_{\mathsf{oe}}{\leftarrow}\; \lambda x.s$.
- *Application right for* $u \;{}_{\mathsf{xe}}{\leftarrow}\; t$ *and application left for* $t \to_{\mathsf{xm}} p$; *i.e.*, $u = qr' \;{}_{\mathsf{xe}}{\leftarrow}\; qr = t$ and $t = qr \to_{\mathsf{xm}} q'r = p$. There are several sub-cases to this:
  – Let $u = qO\langle\tilde{r'}\rangle \;{}_{\mathsf{xe}}{\leftarrow}\; qO\langle\tilde{r}\rangle = t$, with $\tilde{r'} \;{}_{\mathsf{oe}}{\leftarrow}\; \tilde{r}$, and $t = R\langle\tilde{q}\rangle r \to_{\mathsf{xm}} R\langle\tilde{q'}\rangle r = p$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$. Let $t' = R\langle\tilde{q'}\rangle O\langle\tilde{r'}\rangle$, having that
  $$u = R\langle\tilde{q}\rangle O\langle\tilde{r'}\rangle \to_{\mathsf{xm}} t' \;{}_{\mathsf{xe}}{\leftarrow}\; R\langle\tilde{q'}\rangle O\langle\tilde{r}\rangle = p$$
  – Let $u = qO_1\langle\tilde{r'}\rangle \;{}_{\mathsf{xe}}{\leftarrow}\; qO_1\langle\tilde{r}\rangle = t$, and $t = O_2\langle\tilde{q}\rangle r \to_{\mathsf{xm}} O_2\langle\tilde{q'}\rangle r = p$, with $\tilde{q} \to_{\mathsf{xm}} \tilde{q'}$. Then the statement holds by Lemma B.3.3
  – Let $u = qS\langle\tilde{r'}\rangle \;{}_{\mathsf{xe}}{\leftarrow}\; qS\langle\tilde{r}\rangle = t$, with q a rigid term and $\tilde{r'} \;{}_{\mathsf{oe}}{\leftarrow}\; \tilde{r}$, and $t = R\langle\tilde{q}\rangle r \to_{\mathsf{xm}} R\langle\tilde{q'}\rangle r = p$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$. Let $t' = R\langle\tilde{q'}\rangle S\langle\tilde{r'}\rangle$, having that
  $$u = R\langle\tilde{q}\rangle S\langle\tilde{r'}\rangle \to_{\mathsf{xm}} t' \;{}_{\mathsf{xe}}{\leftarrow}\; R\langle\tilde{q'}\rangle S\langle\tilde{r}\rangle = p$$

Note that $t' \ {}_{\mathsf{xe}}\!\leftarrow p$ holds because $R\langle\tilde{q'}\rangle$ is a rigid term —by Lemma C.1.3.

– Let $u = qS\langle\tilde{r'}\rangle \ {}_{\mathsf{xe}}\!\leftarrow qS\langle\tilde{r}\rangle = t$, with q a rigid term and $\tilde{r'} \ {}_{\mathsf{oe}}\!\leftarrow \tilde{r}$, and $t = O\langle\tilde{q}\rangle r \rightarrow_{\mathsf{xm}} O\langle\tilde{q'}\rangle r = p$, with $\tilde{q} \mapsto_{\mathsf{m}} \tilde{q'}$. Let $t' = O\langle\tilde{q'}\rangle S\langle\tilde{r'}\rangle$, having that

$$u = O\langle\tilde{q}\rangle S\langle\tilde{r'}\rangle \rightarrow_{\mathsf{xm}} t' \ {}_{\mathsf{xe}}\!\leftarrow O\langle\tilde{q'}\rangle S\langle\tilde{r}\rangle = p$$

Note that $t' \ {}_{\mathsf{xe}}\!\leftarrow p$ holds because $O\langle\tilde{q'}\rangle$ is rigid—by Lemma C.1.2.

- *Application left for* $u \ {}_{\mathsf{xe}}\!\leftarrow t$ *and application right for* $t \rightarrow_{\mathsf{xm}} p$; *i.e.*, $u = q'r \ {}_{\mathsf{xe}}\!\leftarrow qr = t$ and $t = qr \rightarrow_{\mathsf{xm}} qr' = p$. There are several sub-cases to this:
  – Let $u = R\langle\tilde{q'}\rangle r \ {}_{\mathsf{xe}}\!\leftarrow R\langle\tilde{q}\rangle r = t$, with $\tilde{q'} \ {}_{\mathsf{oe}}\!\leftarrow \tilde{q}$, and $t = qO\langle\tilde{r}\rangle \rightarrow_{\mathsf{xm}} qO\langle\tilde{r'}\rangle = p$, with $\tilde{r} \mapsto_{\mathsf{m}} \tilde{r'}$. Let $t' = R\langle\tilde{q'}\rangle O\langle\tilde{r'}\rangle$, having that
  $$u = R\langle\tilde{q'}\rangle O\langle\tilde{r}\rangle \rightarrow_{\mathsf{xm}} t' \ {}_{\mathsf{xe}}\!\leftarrow R\langle\tilde{q}\rangle O\langle\tilde{r'}\rangle = p$$
  – Let $u = O_1\langle\tilde{q'}\rangle r \ {}_{\mathsf{xe}}\!\leftarrow O_1\langle\tilde{q}\rangle r = t$, with $\tilde{q'} \ {}_{\mathsf{oe}}\!\leftarrow \tilde{q}$, and $t = qO_2\langle\tilde{r}\rangle \rightarrow_{\mathsf{xm}} qO_2\langle\tilde{r'}\rangle = p$, with $\tilde{r} \mapsto_{\mathsf{m}} \tilde{r'}$. Then the statement holds by Lemma B.3.3
  – Let $u = R\langle\tilde{q'}\rangle r \ {}_{\mathsf{xe}}\!\leftarrow R\langle\tilde{q}\rangle r = t$, with $\tilde{q'} \ {}_{\mathsf{oe}}\!\leftarrow \tilde{q}$, and $t = qS\langle\tilde{r}\rangle \rightarrow_{\mathsf{xm}} qS\langle\tilde{r'}\rangle = p$, with $\tilde{r} \rightarrow_{\mathsf{om}} \tilde{r'}$ and q a rigid term. Let $t' = R\langle\tilde{q'}\rangle S\langle\tilde{r'}\rangle$, having that
  $$u = R\langle\tilde{q'}\rangle S\langle\tilde{r}\rangle \rightarrow_{\mathsf{xm}} t' \ {}_{\mathsf{xe}}\!\leftarrow R\langle\tilde{q}\rangle S\langle\tilde{r'}\rangle = p$$
  Note that $u \rightarrow_{\mathsf{xm}} t'$ holds because $R\langle\tilde{q'}\rangle$ is a rigid term —by Lemma C.1.3.
  – Let $u = O\langle\tilde{q'}\rangle r \ {}_{\mathsf{xe}}\!\leftarrow O\langle\tilde{q}\rangle r = t$, with $q' \ {}_{\mathsf{oe}}\!\leftarrow \tilde{q}$, and $t = qS\langle\tilde{r}\rangle \rightarrow_{\mathsf{xm}} qS\langle\tilde{r'}\rangle = p$, with q a rigid term and $\tilde{r} \rightarrow_{\mathsf{om}} \tilde{r'}$. Let $t' = O\langle\tilde{q'}\rangle S\langle\tilde{r'}\rangle$, having that
  $$u = O\langle\tilde{q'}\rangle S\langle\tilde{r}\rangle \rightarrow_{\mathsf{xm}} t' \ {}_{\mathsf{xe}}\!\leftarrow O\langle\tilde{q}\rangle S\langle\tilde{r'}\rangle = p$$
  Note that $u \rightarrow_{\mathsf{xm}} t'$ holds because $O\langle\tilde{q'}\rangle$ is rigid—by Lemma C.1.2.
- *Application right for both* $u \ {}_{\mathsf{xe}}\!\leftarrow t$ *and* $t \rightarrow_{\mathsf{xm}} p$; *i.e.*, $u = qr' \ {}_{\mathsf{xe}}\!\leftarrow qr = t$ and $t = qr \rightarrow_{\mathsf{xm}} qr'' = p$. By *i.h.*, there exists $s \in \Lambda_{\mathsf{vsc}}$ such that $r' \rightarrow_{\mathsf{xm}} s \ {}_{\mathsf{xe}}\!\leftarrow r''$. The analysis of the sub-cases, depending on the open/strong/rigid type contexts involved in $u \ {}_{\mathsf{xe}}\!\leftarrow t$ and $t \rightarrow_{\mathsf{xm}} p$ follows the same schema as for the previous item, all showing that
  $$u = qr' \rightarrow_{\mathsf{xm}} qs \ {}_{\mathsf{xe}}\!\leftarrow qr'' = p$$
- *Application left for both* $u \ {}_{\mathsf{xe}}\!\leftarrow t$ *and* $t \rightarrow_{\mathsf{xm}} p$; *i.e.*, $u = q'r \ {}_{\mathsf{xe}}\!\leftarrow qr = t$ and $t = qr \rightarrow_{\mathsf{xm}} q''r = p$. By *i.h.*, there exists $s \in \Lambda_{\mathsf{vsc}}$ such that $q' \rightarrow_{\mathsf{xm}} s \ {}_{\mathsf{xe}}\!\leftarrow q''$. The analysis of the sub-cases, depending on the open/strong/rigid type contexts involved in $u \ {}_{\mathsf{xe}}\!\leftarrow t$ and $t \rightarrow_{\mathsf{xm}} p$ follows the same schema as for the previous item, all showing that
  $$u = q'r \rightarrow_{\mathsf{xm}} sr \ {}_{\mathsf{xe}}\!\leftarrow q''r = p$$
- ES *right for* $u \ {}_{\mathsf{xe}}\!\leftarrow t$ *and* ES *left for* $t \rightarrow_{\mathsf{xm}} p$; *i.e.*, $u = q[x\leftarrow r'] \ {}_{\mathsf{xe}}\!\leftarrow q[x\leftarrow r] = t$ and $t = q[x\leftarrow r] \rightarrow_{\mathsf{xm}} q'[x\leftarrow r] = p$. There are several sub-cases to this:
  – Let $u = q[x\leftarrow O\langle\tilde{r'}\rangle] \ {}_{\mathsf{xe}}\!\leftarrow q[x\leftarrow O\langle\tilde{r}\rangle] = t$, with $\tilde{r'} \ {}_{\mathsf{e}}\!\leftarrow\!\!\shortmid \tilde{r}$, and $t = O\langle\tilde{q}\rangle[x\leftarrow r] \rightarrow_{\mathsf{xm}} O\langle\tilde{q'}\rangle[x\leftarrow r] = p$, with $\tilde{q} \mapsto_{\mathsf{m}} \tilde{q'}$. Then the statement holds by Lemma B.3.3.
  – Let $u = q[x\leftarrow O\langle\tilde{r'}\rangle] \ {}_{\mathsf{xe}}\!\leftarrow q[x\leftarrow O\langle\tilde{r}\rangle] = t$, with $\tilde{r'} \ {}_{\mathsf{e}}\!\leftarrow\!\!\shortmid \tilde{r}$, and $t = S\langle\tilde{q}\rangle[x\leftarrow r] \rightarrow_{\mathsf{xm}} S\langle\tilde{q'}\rangle[x\leftarrow r] = p$, with $\tilde{q} \rightarrow_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let $t' \coloneqq S\langle\tilde{q'}\rangle[x\leftarrow O\langle\tilde{r'}\rangle]$, having that
  $$u = S\langle\tilde{q}\rangle[x\leftarrow O\langle\tilde{r'}\rangle] \rightarrow_{\mathsf{xm}} t' \rightarrow_{\mathsf{xm}} S\langle\tilde{q'}\rangle[x\leftarrow O\langle\tilde{r}\rangle] = p$$
  Note that $u \rightarrow_{\mathsf{xm}} t'$ holds because $O\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.2
  – Let $u = q[x\leftarrow O\langle\tilde{r'}\rangle] \ {}_{\mathsf{xe}}\!\leftarrow q[x\leftarrow O\langle\tilde{r}\rangle] = t$, with $\tilde{r'} \ {}_{\mathsf{e}}\!\leftarrow\!\!\shortmid \tilde{r}$, and $t = R\langle\tilde{q}\rangle[x\leftarrow r] \rightarrow_{\mathsf{xm}} R\langle\tilde{q'}\rangle[x\leftarrow r] = p$, with $\tilde{q} \rightarrow_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let $t' \coloneqq R\langle\tilde{q'}\rangle[x\leftarrow O\langle\tilde{r'}\rangle]$, having that
  $$u = R\langle\tilde{q}\rangle[x\leftarrow O\langle\tilde{r'}\rangle] \rightarrow_{\mathsf{xm}} t' \rightarrow_{\mathsf{xm}} R\langle\tilde{q'}\rangle[x\leftarrow O\langle\tilde{r}\rangle] = p$$
  Note that $u \rightarrow_{\mathsf{xm}} t'$ holds because $O\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.2
  – Let $u = q[x\leftarrow R\langle\tilde{r'}\rangle] \ {}_{\mathsf{xe}}\!\leftarrow q[x\leftarrow R\langle\tilde{r}\rangle] = t$, with $r' \ {}_{\mathsf{oe}}\!\leftarrow r$, and $t = O\langle\tilde{q}\rangle[x\leftarrow r] \rightarrow_{\mathsf{xm}} O\langle\tilde{q'}\rangle[x\leftarrow r] = p$, with $\tilde{r} \rightarrow_{\mathsf{xm}} \tilde{r'}$. Let $t' \coloneqq O\langle\tilde{q'}\rangle[x\leftarrow R\langle r'\rangle]$, having that
  $$u = O\langle\tilde{q}\rangle[x\leftarrow R\langle\tilde{r'}\rangle] \rightarrow_{\mathsf{xm}} t' \ {}_{\mathsf{xe}}\!\leftarrow O\langle\tilde{q'}\rangle[x\leftarrow R\langle\tilde{r}\rangle] = p$$
  – Let $u = q[x\leftarrow R\langle\tilde{r'}\rangle] \ {}_{\mathsf{xe}}\!\leftarrow q[x\leftarrow R\langle\tilde{r}\rangle] = t$, with $\tilde{r'} \ {}_{\mathsf{oe}}\!\leftarrow \tilde{r}$, and $t = S\langle\tilde{q}\rangle[x\leftarrow r] \rightarrow_{\mathsf{xm}} S\langle\tilde{q'}\rangle[x\leftarrow r] = p$, with $\tilde{q} \rightarrow_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let $t' \coloneqq S\langle\tilde{q'}\rangle[x\leftarrow R\langle\tilde{r'}\rangle]$, having that
  $$u = S\langle\tilde{q}\rangle[x\leftarrow R\langle\tilde{r'}\rangle] \rightarrow_{\mathsf{xm}} t' \ {}_{\mathsf{xe}}\!\leftarrow S\langle\tilde{q'}\rangle[x\leftarrow R\langle\tilde{r}\rangle] = p$$

Note that $u \to_{\mathsf{xm}} t'$ holds because $R\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.3.

– Let $u = q[x{\leftarrow}R_1\langle\tilde{r'}\rangle] \ {}_{\mathsf{xe}}{\leftarrow}\ q[x{\leftarrow}R_1\langle\tilde{r}\rangle] = t$, with $\tilde{r'} \ {}_{\mathsf{oe}}{\leftarrow}\ \tilde{r}$, and $t = R_2\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} R_2\langle\tilde{q'}\rangle[x{\leftarrow}r]$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let

$$t' \coloneqq R_2\langle\tilde{q'}\rangle[x{\leftarrow}R_1\langle\tilde{r'}\rangle]$$

having that

$$\begin{array}{rcl} u & = & R_2\langle\tilde{q}\rangle[x{\leftarrow}R_1\langle\tilde{r'}\rangle] \\ & \to_{\mathsf{xm}} & t' \ {}_{\mathsf{xe}}{\leftarrow}\ R_2\langle\tilde{q'}\rangle[x{\leftarrow}R_1\langle\tilde{r}\rangle] \end{array}$$

Note that $u \to_{\mathsf{xm}} t'$ holds because $R_1\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.3.

– Let $u = q[x{\leftarrow}R\langle\tilde{r'}\rangle] \ {}_{\mathsf{xe}}{\leftarrow}\ q[x{\leftarrow}R\langle\tilde{r}\rangle] = t$, with $\tilde{r'} \ {}_{\mathsf{oe}}{\leftarrow}\ \tilde{r}$ and q is a rigid term, and $t = O\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} O\langle\tilde{q'}\rangle[x{\leftarrow}r] = p$, with $\tilde{q} \mapsto_{\mathsf{m}} \tilde{q'}$. Let $t' \coloneqq O\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle]$, having that

$$u = O\langle\tilde{q}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle] \to_{\mathsf{xm}} t' \ {}_{\mathsf{xe}}{\leftarrow}\ O\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r}\rangle] = p$$

Note that $t' \ {}_{\mathsf{xe}}{\leftarrow}\ p$ holds because $O\langle\tilde{q'}\rangle$ is a rigid term —by Lemma C.1.2.

– Let $t = q[x{\leftarrow}R\langle\tilde{r'}\rangle] \ {}_{\mathsf{xe}}{\leftarrow}\ q[x{\leftarrow}R\langle\tilde{r}\rangle] = t$, with $\tilde{r'} \ {}_{\mathsf{oe}}{\leftarrow}\ \tilde{r}$ and q is a rigid term, and $t = S\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} S\langle\tilde{q'}\rangle[x{\leftarrow}r]$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let

$$t' \coloneqq S\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle]$$

having that

$$u = S\langle\tilde{q}\rangle[x{\leftarrow}R\langle\tilde{r'}\rangle] \to_{\mathsf{xm}} t' \ {}_{\mathsf{xe}}{\leftarrow}\ S\langle\tilde{q'}\rangle[x{\leftarrow}R\langle\tilde{r}\rangle] = p$$

Note that $u \to_{\mathsf{xm}} t'$ holds because $R\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.3—, and that $t \ {}_{\mathsf{xe}}{\leftarrow}\ p$ holds because $S\langle\tilde{q'}\rangle$ is a rigid term —by Lemma C.1.3.

– Let $u = q[x{\leftarrow}R_1\langle\tilde{r'}\rangle] \ {}_{\mathsf{xe}}{\leftarrow}\ q[x{\leftarrow}R_1\langle\tilde{r}\rangle] = t$, with $\tilde{r'} \ {}_{\mathsf{oe}}{\leftarrow}\ \tilde{r}$ and q is a rigid term, and $t = R_2\langle\tilde{q}\rangle[x{\leftarrow}r] \to_{\mathsf{xm}} R_2\langle\tilde{q'}\rangle[x{\leftarrow}r] = p$, with $\tilde{q} \to_{\mathsf{om}} \tilde{q'}$ and r is a rigid term. Let $t' \coloneqq R_2\langle\tilde{q'}\rangle[x{\leftarrow}R_1\langle\tilde{r'}\rangle]$, having that

$$uR_2\langle\tilde{q}\rangle[x{\leftarrow}R_1\langle\tilde{r'}\rangle] \to_{\mathsf{xm}} t' \ {}_{\mathsf{xe}}{\leftarrow}\ R_2\langle\tilde{q'}\rangle[x{\leftarrow}R_1\langle\tilde{r}\rangle] = p$$

Note that $u \to_{\mathsf{xm}} t'$ holds because $R_1\langle\tilde{r'}\rangle$ is a rigid term —by Lemma C.1.3—, and that $t' \ {}_{\mathsf{xe}}{\leftarrow}\ p$ because $R_2\langle\tilde{q'}\rangle$ is a rigid term —by Lemma C.1.3.

- ES *left for* $u \ {}_{\mathsf{xe}}{\leftarrow}\ t$ *and* ES *right for* $t \to_{\mathsf{xm}} p$; *i.e.*, $u = q'[x{\leftarrow}r] \ {}_{\mathsf{xe}}{\leftarrow}\ q[x{\leftarrow}r] = t$ and $t = q[x{\leftarrow}r] \to_{\mathsf{xm}} q[x{\leftarrow}r'] = p$. There are several sub-cases to this, all of which follow the same kind of reasoning as for the case ES *right for* $u \ {}_{\mathsf{xe}}{\leftarrow}\ t$ *and* ES *left for* $t \to_{\mathsf{xm}} p$. Therefore, we shall leave this case for the reader.
- ES *right for both* $u \ {}_{\mathsf{xe}}{\leftarrow}\ t$ *and* $t \to_{\mathsf{xm}} p$; *i.e.*,

$$u = q[x{\leftarrow}r'] \ {}_{\mathsf{xe}}{\leftarrow}\ q[x{\leftarrow}r] = t$$

and

$$t = q[x{\leftarrow}r] \to_{\mathsf{xm}} q[x{\leftarrow}r''] = p$$

By *i.h.* there exists $s \in \Lambda_{\mathsf{vsc}}$ such that $r' \to_{\mathsf{xm}} r \ {}_{\mathsf{xe}}{\leftarrow}\ r''$. The analysis of the sub-cases, depending on the open/strong/rigid type contexts involved in $t \to_{\mathsf{xm}} u$ and $t \to_{\mathsf{xm}} p$, follows the same schema as for the previous item, all showing that

$$u = q[x{\leftarrow}r'] \to_{\mathsf{xm}} q[x{\leftarrow}s] \ {}_{\mathsf{xe}}{\leftarrow}\ q[x{\leftarrow}r''] = p$$

- ES *left for both* $u \ {}_{\mathsf{xe}}{\leftarrow}\ t$ *and* $t \to_{\mathsf{xm}} p$; *i.e.*,

$$u = q'[x{\leftarrow}r] \ {}_{\mathsf{xe}}{\leftarrow}\ q[x{\leftarrow}r] = t$$

and

$$t = q[x{\leftarrow}r] \to_{\mathsf{xm}} q''[x{\leftarrow}r] = p$$

By *i.h.* there exists $s \in \Lambda_{\mathsf{vsc}}$ such that $r' \to_{\mathsf{xm}} r \ {}_{\mathsf{xe}}{\leftarrow}\ r''$. The analysis of the sub-cases, depending on the open / strong / rigid type contexts involved in $t \to_{\mathsf{xm}} u$ and $t \to_{\mathsf{xm}} p$, follows the same schema as for the previous item, all showing that

$$u = q'[x{\leftarrow}r] \to_{\mathsf{xm}} s[x{\leftarrow}r] \ {}_{\mathsf{xe}}{\leftarrow}\ q''[x{\leftarrow}r] = p. \quad \square$$

Proposition C.3 (Properties of $\to_{\mathsf{x}}$)**.** *Let t be a VSC term.*

1) Diamond*: $\to_{\mathsf{x}}$ is diamond. Moreover, every $\to_{\mathsf{x}}$ evaluation to normal form (if any) has the same number of $\to_{\mathsf{xm}}$ steps.*

2) Normal forms*: if t is $\to_{\mathsf{x}}$ normal then it is a strong fireball.*

See p. 5 Prop. III.1

In the following proof we refer to terms of the form $L\langle t\rangle$ as *answers*.

Proof. 1) Follows from strong commutation of $\to_{\mathsf{xm}}$ and $\to_{\mathsf{xe}}$ (Lemma C.2.2), from diamond for $\to_{\mathsf{xm}}$ and $\to_{\mathsf{xe}}$ (Lemma C.2.1), and from Hindley-Rosen lemma ([20, Prop. 3.3.5]). By the random descent property (which is a well-known corollary of the diamond recalled in Sect. II), all $\to_{\mathsf{x}}$ evaluation sequences to normal form have the same number of steps. By strong commutation of $\to_{\mathsf{xm}}$ and $\to_{\mathsf{xe}}$, they also have the number of $\to_{\mathsf{xm}}$ steps.

2) To have the right *i.h.*, we prove simultaneously, by induction on t, the following stronger statements (we recall that all strong inert terms are strong fireballs):
   a) *Fireball property*: If t is x-normal, then t is a strong fireball.
   b) *Non-value property*: If t is x-normal and not an answer, then t is a strong inert term.

   Cases:
   - *Variable*, *i.e.*, $t = x$: both properties trivially hold, since t is a strong inert term and so a strong fireball.
   - *Abstraction*, *i.e.*, $t = \lambda x.u$:
     a) *Non-value property*: vacuously true, as t is an abstraction and hence an answer.
     b) *Fireball property*: Since t is x-normal, so is u. By *i.h.* applied to u (fireball property), u is a strong fireball and hence so is t (as a strong value).
   - *Application*; *i.e.*, $t = t_1 t_2$ (which is not an answer):
     a) *Non-value property*: Since t is x-normal, so are t_1 and t_2. Moreover, t_1 is not an answer (otherwise t would be a $\to_{\mathsf{xm}}$-redex). By *i.h.* applied to t_1 (non-value property) and to t_2 (fireball property), t_1 is a strong inert term and t_2 is a strong fireball. Thus, t is a strong inert term.
     b) *Fireball property*: We have just proved that t is a strong inert term, and hence it is a strong fireball.
   - *Explicit substitutions*, *i.e.*, $t = t_1[x \leftarrow t_2]$:
     a) *Fireball property*: Since t is x-normal, so are t_1 and t_2. Moreover, t_2 is not an answer (otherwise t would be a $\to_{\mathsf{xe}_\lambda}$-redex). By *i.h.* applied to t_1 (fireball property) and to t_2 (non-value property), t_1 is a strong fireball and t_2 is a strong inert term. Thus, t is a strong fireball.
     b) *Non-value property*: We have just proved that t is a strong fireball. If moreover t is not a answer, then t_1 is not an answer and hence, by *i.h.* applied to t_1 (non-value property), t_1 is a strong inert term. Therefore, t is a strong inert term. □

Proposition C.4 ($\equiv$ is a strong bisimulation)**.** *If $t \equiv u$ and $t \to_{\mathsf{a}} t'$ then there exists $u' \in \Lambda_{\mathsf{vsc}}$ such that $u \to_{\mathsf{a}} u'$ and $t' \equiv u'$, for $\mathsf{a} \in \{\mathsf{m}, \mathsf{e}, \mathsf{om}, \mathsf{oe}, \mathsf{xm}, \mathsf{xe}\}$.* See p. 5 Prop. III.2

Proof. Easy adaptation of the proof in [11, Lemma 12]. □

APPENDIX D
PROOFS OF SECTION V (RELAXED IMPLEMENTATION)

This section proves Thm. V.3 (proved in Thm. D.2) using the following auxiliary lemma.

Lemma D.1 (One-step transfer)**.** *Let M and $(\to, \equiv)$ be a relaxed implementation system. For any state s of M, if $s{\downarrow} \to u$ then there is a state s' of M such that $s \rightsquigarrow_{\mathsf{o}}^* \rightsquigarrow_\beta s'$.*

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

Theorem D.2 (Sufficient condition for implementations)**.** *Let M and $(\to, \equiv)$ be a relaxed implementation system. Then, M is a relaxed implementation of $(\to, \equiv)$.* See p. 7 Thm. V.3

Proof. 1) *Executions to evaluations*: by induction on the length of the execution. It follows easily from relaxed β-projection and overhead transparency, plus the strong bisimulation of $\equiv$ with respect to $\to$.

2) *Normalizing evaluations to executions*: we prove a more general statement where we replace t° with a general state s, and t with $s{\downarrow}$: *if $d : s{\downarrow} \to^* u$ with u normal form then there exists an M-execution $\rho : s \rightsquigarrow^* s'$ with s' final such that $s'{\downarrow} \equiv u$ with $|\rho|_\beta \le |d|_{\mathsf{m}}$*. Then if we instantiate this more general statement on $s = t^\circ$ we obtain the official statement, because by the initialization constraint of the machine we have $t^\circ{\downarrow} = t$.
   The proof of the generalized statement is by induction on $|d|_{\mathsf{m}}$. If $|d|_{\mathsf{m}} = 0$ then consider $\mathsf{nf}_{\mathsf{o}}(s)$, that by overhead transparency (Point 2) satisfies $\mathsf{nf}_{\mathsf{o}}(s){\downarrow} = s{\downarrow}$. Now, if $\mathsf{nf}_{\mathsf{o}}(s)$ has a β-transition then from $s{\downarrow}$ it is eventually possible to do a $\to_{\mathsf{m}}$ steps by relaxed β-projection (Point 1), which is impossible, because $|d|_{\mathsf{m}} = 0$ and by the diamond property all

evaluations sequences from a term have the same number and kind of steps. Then, $\mathsf{nf}_{\mathsf{o}}(s)$ is a final state and there is an execution $\rho : s \rightsquigarrow_{\mathsf{o}}^{*} \mathsf{nf}_{\mathsf{o}}(s)$ such that $|\rho|_{\beta} = 0$.

If $|d|_{\mathsf{m}} > 0$ then $s{\downarrow} \to^{+} u$ and $s{\downarrow} \to p \to^{*} u$ for some p. By the one-step transfer lemma (Lemma D.1), we obtain $s \rightsquigarrow_{\mathsf{o}}^{*} \rightsquigarrow_{\beta} s'$ for some s'. By overhead transparency and relaxed β projection, we obtain $d' : s{\downarrow} \to^{+} q \equiv s'{\downarrow}$ for some q and with $|d'|_{\mathsf{m}} \geq 1$. By diamond of $\to$, we obtain an evaluation $e : q \to^{*} u$ such that $|e|_{\mathsf{m}} \leq |d|_{\mathsf{m}} - 1$. Note also that by strong bisimulation of $\equiv$ applied to e we obtain $e' : s'{\downarrow} \to^{|e|} u'$ with $u' \equiv u$. We can then apply the *i.h.*, obtaining an execution $\sigma : s' \rightsquigarrow^{*} s''$ with s'' final and such that $s''{\downarrow} \equiv u'$, and $|\sigma|_{\beta} \leq |e|_{\mathsf{m}} \leq |d|_{\mathsf{m}} - 1$. Note that the execution $\rho : s \rightsquigarrow_{\mathsf{o}}^{*} \rightsquigarrow_{\beta} s' \rightsquigarrow^{*} s''$ satisfies the statement because $s''{\downarrow} \equiv u' \equiv u$ and $|\rho|_{\beta} = |\sigma|_{\beta} + 1 \leq |e|_{\mathsf{m}} + 1 \leq |d|_{\mathsf{m}}$.

3) *Diverging evaluations to executions*: suppose that $\to$ diverges on t but M terminates, that is, that there is an execution $\rho : t^{\circ} \rightsquigarrow^{*} s$ with s final. Then the projection $\rho{\downarrow} : t \to^{*} u \equiv s{\downarrow}$ for some u given by point 1 (of this theorem) is a normalizing sequence by the halt property (guaranteeing that $s{\downarrow}$ is $\to$ normal), and by strong bisimulation so is the evaluation $t \to^{*} u$. Then $\to$ normalizes t and so, by diamond (precisely by *uniform normalization* implied by the diamond property, see footnote Sect. II), $\to$ cannot diverge on t—absurd. Therefore M diverges on t°. Now, since $\rightsquigarrow_{\mathsf{o}}$ terminates, the diverging execution from t° must have infinitely many β transitions. Note that a diverging $\to$ sequence necessarily has an infinity of $\to_{\mathsf{m}}$-steps, because $\to_{\mathsf{e}}$ terminates (Lemma II.1). □

APPENDIX E
PROOFS OF SECTION VII (COMPILATION & READ-BACK)

In this section, we introduce formally the notions of *well-namedness*, crumbling, and read-back. The main results are some fundamental properties about the crumbling translation – Lemma VII.4 (proved in Lemma E.22) – and the modular read-back of crumbled terms – Lemma VII.6 (proved in Lemma E.27). To achieve the goal we will also provide first a number of additional technical properties on (well-named) environments and on the definitions of read-back.

A. λ-terms & well-namedness

Definition E.1 (Capture-avoiding substitution)**.** *We denote by $t\{x{\leftarrow}u\}$ the term obtained by replacing the variable x with u in t. The operation of replacing may rename bound variables in order to avoid captures, but we assume that it minimizes the number of renamings by performing only the strictly necessary ones.*

Definition E.2. *A* substitution *σ is a mapping from variables to terms such that it is the identity on all but a finite number of variables. Its* domain $\mathsf{dom}(\sigma)$ *is the finite set of variables that are not mapped to themselves. The set* $\mathsf{fv}(\sigma)$ *of free variables of σ is the set of all variables that occur free in at least one $\sigma(y)$ for $y \in \mathsf{dom}(\sigma)$. It is a* fireball substitution *if it maps variables to fireballs. We write $\sigma_1\sigma_2$ for the composed substitution defined by $(\sigma_1\sigma_2)(x) \coloneqq \sigma_2(\sigma_1(x))$. We write $\{x{\leftarrow}t\}$ for a substitution on a single variable and $t\sigma$ for the term obtained from t by the capture-avoiding replacement of every $x \in \mathsf{fv}(t)$ with σx. Similarly for every other syntactic category, e.g. $e\sigma$.*

The terms that we operate on are subject to well-namedness, which basically amounts to Barendregt's variable convention. To lighten up the proofs, we introduce the following shorthand to denote disjoint sets of variables:

Definition E.3 (Disjoint variables – λ-calculus version)**.** *Let X and Y be sets of variable names. We say that X and Y are disjoint (in symbols, $X \perp Y$) if $X \cap Y = \emptyset$.*

Definition E.4 (Well-named λ-terms)**.** *A λ-term t is* well-named *if its bound variables are all distinct, and $\mathsf{fv}(t) \perp \mathsf{bv}(t)$.*

B. Crumbled environments

Definition E.5 (Capture-avoiding substitution)**.** *We denote by $e\{x{\leftarrow}y\}$ and $b\{x{\leftarrow}y\}$ the environment (resp. bite) obtained by replacing the variable x with y in e (resp. b). The operation of replacing may rename bound variables in order to avoid captures, but we assume that it minimizes the number of renamings by performing only the strictly necessary ones.*

Definition E.6 (Free and bound variables)**.** *We define the sets of* free *and* bound variables *of crumbled environments and bites, and the* domain *of crumbled environments, as expected:*

$$\begin{array}{cc} \mathsf{bv}(\epsilon) \coloneqq \emptyset & \mathsf{bv}(e[x{\leftarrow}b]) \coloneqq \mathsf{bv}(e) \cup \{x\} \cup \mathsf{bv}(b) \\ \mathsf{fv}(\epsilon) \coloneqq \{\star\} & \mathsf{fv}(e[x{\leftarrow}b]) \coloneqq \mathsf{fv}(e) \setminus \{x\} \cup \mathsf{fv}(b) \\ \mathsf{dom}(\epsilon) \coloneqq \emptyset & \mathsf{dom}(e[x{\leftarrow}b]) \coloneqq \mathsf{dom}(e) \cup \{x\} \end{array}$$

$$\begin{array}{ccc} \mathsf{bv}(x) \coloneqq \emptyset & \mathsf{bv}(xy) \coloneqq \emptyset & \mathsf{bv}(\lambda x.e) \coloneqq \{x\} \cup \mathsf{bv}(e) \\ \mathsf{fv}(x) \coloneqq \{x\} & \mathsf{fv}(xy) \coloneqq \{x, y\} & \mathsf{fv}(\lambda x.e) \coloneqq \mathsf{fv}(e) \setminus \{x\} \end{array}$$

Moreover, $\mathsf{vars}(e) \coloneqq \mathsf{fv}(e) \cup \mathsf{bv}(e)$ *and* $\mathsf{vars}(b) \coloneqq \mathsf{fv}(b) \cup \mathsf{bv}(b)$.

Definition E.7 (α-equality $=_\alpha$)**.** *We define the relation $=_\alpha$ as the smallest equivalence relation (reflexive, symmetric, transitive) over crumbled forms, satisfying the following properties:*

1) Structural, ES: *If $e =_\alpha e'$ and $b =_\alpha b'$ then $e[x\leftarrow b] =_\alpha e'[x\leftarrow b']$.*
2) Rename, ES: $e[x\leftarrow b] =_\alpha e\{x\leftarrow y\}[y\leftarrow b]$ *when* $y \notin \mathsf{vars}(e)$.
3) Rename, abstraction: $\lambda x.e =_\alpha \lambda y.(e\{x\leftarrow y\})$ *when* $y \notin \mathsf{vars}(e)$.
4) Structural, abstraction: *If $e =_\alpha e'$ then $\lambda x.e =_\alpha \lambda x.e'$.*

Moreover, in points 2 and 3, α-equivalence can rename x with y only if they are both in $V_{\mathsf{cr}} \setminus \{\star\}$ (resp. both in V_{calc}); $\star$ cannot be renamed.

Given an environment e, we denote by e^α any environment that is α-equivalent to e: we call e^α a "copy" of e. Later, when necessary, we will attach additional requirements on the variables of e^α, for instance that $\mathsf{bv}(e^\alpha)$ is disjoint from a certain set of variables.

In crumbled environments may occur so-called crumbling variables (V_{cr}). We do not handle these variables any different with respect to well-namedness: the only difference is that we ignore the special variable $\star$, both when defining disjoint set of variables, and in well-namedness.

Definition E.8 (Disjoint variables – crumbled version)**.** *Let X and Y be sets of variable names, i.e. $X, Y \subseteq V_{\mathsf{calc}} \cup V_{\mathsf{cr}}$. We say that X and Y are disjoint (in symbols, $X \perp Y$) if $X \cap Y \subseteq \{\star\}$.*

Definition E.9 (Well-named crumbled forms)**.** *An environment e (resp. a bite b) is* well-named *if its bound variables (not including $\star$) are all distinct, and $\mathsf{fv}(e) \perp \mathsf{bv}(e)$ (resp. $\mathsf{fv}(b) \perp \mathsf{bv}(b)$).*

Lemma E.10 (Variables and concatenation)**.** *For all crumbled environments e and e':*

- $\mathsf{fv}(ee') \subseteq \mathsf{fv}(e) \setminus \mathsf{dom}(e') \cup \mathsf{fv}(e')$
- $\mathsf{bv}(ee') = \mathsf{bv}(e) \cup \mathsf{bv}(e')$
- $\mathsf{vars}(ee') = \mathsf{vars}(e) \cup \mathsf{vars}(e')$
- $\mathsf{dom}(ee') = \mathsf{dom}(e) \cup \mathsf{dom}(e')$.

Proof. Easy, by induction on the structure of e'. □

We provide a formal definition of α-equality for crumbled environments.

Definition E.11 (α-equality $=_\alpha$)**.** *We define the relation $=_\alpha$ as the smallest equivalence relation (reflexive, symmetric, transitive) over terms, satisfying the following properties:*

1) Structural, ES: *If $t =_\alpha t'$ and $u =_\alpha u'$ then $t[x\leftarrow u] =_\alpha t'[x\leftarrow u']$.*
2) Rename, ES: $t[x\leftarrow u] =_\alpha t\{x\leftarrow y\}[y\leftarrow u]$ *when* $y \notin \mathsf{vars}(t)$.
3) Rename, abstraction: $\lambda x.t =_\alpha \lambda y.(t\{x\leftarrow y\})$ *when* $y \notin \mathsf{vars}(t)$.
4) Structural, abstraction: *If $t =_\alpha u$ then $\lambda x.t =_\alpha \lambda x.u$.*

Free variables of a crumbled environment are stable under α-renaming:

Lemma E.12 (Alpha and free variables)**.** *For every environments e, e': if $e =_\alpha e'$ then $\mathsf{fv}(e) = \mathsf{fv}(e')$.*

Proof. Easy, by structural induction on the derivation of $e =_\alpha e'$. □

C. Properties of well-named environments

Lemma E.13 (Concatenation of well-named environments)**.** *Let e, e' be well-named crumbled environments such that:*

- $\mathsf{bv}(e) \perp \mathsf{vars}(e')$,
- $\mathsf{fv}(e) \setminus \mathsf{dom}(e') \perp \mathsf{bv}(e')$.

Then ee' is well-named.

Proof. By induction on the structure of e':

- If $e' = \epsilon$, then $ee' = e$ and we conclude.
- If $e' = e''[x\leftarrow b]$, in order to apply the *i.h.* obtaining that ee'' is well-named, we need the following properties:
  - $\mathsf{bv}(e) \perp \mathsf{vars}(e'')$. Follows from the hypothesis $\mathsf{bv}(e) \perp \mathsf{vars}(e')$ because $\mathsf{vars}(e'') \subseteq \mathsf{vars}(e')$.
  - $\mathsf{fv}(e) \setminus \mathsf{dom}(e'') \perp \mathsf{bv}(e'')$. Note that $\mathsf{dom}(e'') = \mathsf{dom}(e) \setminus \{z\}$, hence $\mathsf{fv}(e) \setminus \mathsf{dom}(e'') \subseteq \mathsf{fv}(e) \setminus \mathsf{dom}(e') \cup \{z\}$. Also, $\mathsf{bv}(e'') \subseteq \mathsf{bv}(e')$. Finally, we use the fact that $z \notin \mathsf{bv}(e'')$, which follows from the hypothesis that $e''[x\leftarrow b]$ is well-named.

  We have just obtained that ee'' is well-named. We need to show that $ee''[x\leftarrow b]$ is well-named.

- Bound variables are all distinct. It suffices to show that $\mathsf{bv}(ee'') \cup \{x\} \perp \mathsf{bv}(b)$, and that $x \notin \mathsf{bv}(ee'')$. Both follow from $\mathsf{bv}(e) \perp \mathsf{vars}(e')$ and from the well-namedness of e'.
- Free variables are distinct from bound ones. By definition, $\mathsf{fv}(ee''[x{\leftarrow}b]) = \mathsf{fv}(ee'') \setminus \{x\} \cup \mathsf{fv}(b)$ and $\mathsf{bv}(ee''[x{\leftarrow}b]) = \mathsf{bv}(ee'') \cup \{x\} \cup \mathsf{bv}(b)$.
  - $\mathsf{fv}(ee'') \setminus \{x\} \perp \mathsf{bv}(ee'') \cup \{x\}$ because $\mathsf{fv}(ee'') \perp \mathsf{bv}(ee'')$ by the well-namedness of ee'';
  - $\mathsf{fv}(b) \perp \mathsf{bv}(b)$ by well-namedness of e';
  - $x \notin \mathsf{fv}(b)$ by well-namedness of e';
  - $\mathsf{fv}(b) \perp \mathsf{bv}(ee'')$ if and only if $\mathsf{fv}(b) \perp \mathsf{bv}(e)$ and $\mathsf{fv}(b) \perp \mathsf{bv}(e'')$. The first follows from $\mathsf{bv}(e) \perp \mathsf{vars}(e')$, the second by well-namedness of e'.
  - $\mathsf{bv}(b) \perp \mathsf{fv}(ee'') \setminus \{x\}$. Note that $x \notin \mathsf{bv}(b)$ by well-namedness of e', therefore we prove instead the equivalent $\mathsf{bv}(b) \perp \mathsf{fv}(ee'')$. The hypothesis $\mathsf{fv}(e) \setminus \mathsf{dom}(e') \perp \mathsf{bv}(e')$ implies $\mathsf{fv}(e) \setminus \mathsf{dom}(e') \perp \mathsf{bv}(b)$, and since by well-namedness of e' $\mathsf{bv}(b) \perp \mathsf{dom}(e')$, $\mathsf{fv}(e) \setminus \mathsf{dom}(e') \perp \mathsf{bv}(b)$ is equivalent to $\mathsf{fv}(e) \perp \mathsf{bv}(b)$. Again by well-namedness of e', $\mathsf{fv}(e'') \perp \mathsf{bv}(b)$, and we conclude because $\mathsf{fv}(ee'') \subseteq \mathsf{fv}(e) \cup \mathsf{fv}(e'')$ by Lemma E.10. □

An auxiliary lemma that will be used in later sections:

Lemma E.14. *If $e[x{\leftarrow}b]$ is well-named then e is well-named and $\mathsf{fv}(b) \perp \mathsf{bv}(e)$.*

Proof. Easy by the definition of well-named. □

D. Read-back properties

Before proving some important properties of the read-back in Lemma E.18, we need the following additional properties of VSC terms.

Lemma E.15 (Alpha & substitution)**.** *If $t =_\alpha t'$ and $u =_\alpha u'$, then $t\{x{\leftarrow}u\} =_\alpha t'\{x{\leftarrow}u'\}$.*

Proof. Easy by structural induction on the derivation of $t =_\alpha t'$, and by the definition of substitution. □

Lemma E.16 (Variables & substitution)**.** *Let t, u be terms. Then:*

1) $\mathsf{fv}(t\{x{\leftarrow}u\}) \subseteq \mathsf{fv}(t) \setminus \{x\} \cup \mathsf{fv}(u)$.
2) $\mathsf{bv}(t\{x{\leftarrow}u\}) \subseteq \mathsf{bv}(t) \cup \mathsf{bv}(u)$ *when* $\mathsf{bv}(t) \perp \mathsf{fv}(u)$.

Proof. Easy, by induction on the structure of t. □

Similar properties hold for crumbled environments:

Lemma E.17. *Let e be an environment such that $y \notin \mathsf{bv}(e)$. Then:*

1) $\mathsf{fv}(e\{x{\leftarrow}y\}) \subseteq \mathsf{fv}(e) \setminus \{x\} \cup \{y\}$.
2) $\mathsf{bv}(e\{x{\leftarrow}y\}) = \mathsf{bv}(e)$.

Proof. Easy, by induction on the structure of e. □

Lemma E.18 (Properties of the read-back)**.** *For all environments e, e':*

1) α-Equality: *if $e =_\alpha e'$ and e, e' are well-named, then $e{\downarrow} =_\alpha e'{\downarrow}$.*
2) Renaming: $e\{x{\leftarrow}y\}{\downarrow} = e{\downarrow}\{x{\leftarrow}y\}$ *when* $y \notin \mathsf{bv}(e)$.
3) Free variables: $\mathsf{fv}(e{\downarrow}) \subseteq \mathsf{fv}(e)$.
4) Bound variables: $\mathsf{bv}(e{\downarrow}) \subseteq \mathsf{bv}(e)$ *when e is well-named.*
5) Dissociation: *if* $\mathsf{fv}(e') \perp \mathsf{dom}(e)$ *and* $\mathsf{dom}(e) \subset V_{\mathsf{cr}}$ *then* $e'[x{\leftarrow}b]e{\downarrow} = e'{\downarrow}\{x{\leftarrow}[\star{\leftarrow}b]e{\downarrow}\}$.

Proof. We prove the points by mutual induction on the size of e:

1) We prove at the same time the corresponding statement for bites, *i.e.* that $b =_\alpha b'$ implies $b{\downarrow} =_\alpha b'{\downarrow}$ for all bites b, b'. By structural induction on the derivation of respectively $e =_\alpha e'$ or $b =_\alpha b'$:
   - *Reflexivity:* in this case $e = e'$ and $b = b'$, and therefore trivially $e{\downarrow} = e'{\downarrow}$ and $b{\downarrow} = b'{\downarrow}$.
   - *Symmetry:* $e =_\alpha e'$ because $e' =_\alpha e$. Then by *i.h.* we obtain $e'{\downarrow} =_\alpha e{\downarrow}$, and we conclude because $=_\alpha$ is symmetric. Similarly for bites.
   - *Transitivity:* $e =_\alpha e'$ because $e =_\alpha e''$ and $e'' =_\alpha e'$. Just use the *i.h.* and use symmetry of $=_\alpha$. Similarly for bites.
   - *Structural, ES:* $e =_\alpha e'$ because $e = e''[x{\leftarrow}b]$, $e' = e'''[x{\leftarrow}b']$ where $e'' =_\alpha e'''$ and $b =_\alpha b'$. Then $e{\downarrow} = e''[x{\leftarrow}b]{\downarrow}$, and there are two subcases:
     - $x \in V_{\mathsf{cr}}$ or b abstraction: then $e''[x{\leftarrow}b]{\downarrow} = e''{\downarrow}\{x{\leftarrow}b{\downarrow}\}$ and $e'''[x{\leftarrow}b']{\downarrow} = e'''{\downarrow}\{x{\leftarrow}b'{\downarrow}\}$. By *i.h.* $e''{\downarrow} =_\alpha e'''{\downarrow}$ and $b{\downarrow} =_\alpha b'{\downarrow}$. We conclude by Lemma E.15.

- $x \in V_{\mathsf{calc}}$: then $e''[x{\leftarrow}b]{\downarrow} = e''{\downarrow}[x{\leftarrow}b{\downarrow}]$ and $e'''[x{\leftarrow}b']{\downarrow} = e'''{\downarrow}[x{\leftarrow}b'{\downarrow}]$. By *i.h.* $e''{\downarrow} =_\alpha e'''{\downarrow}$ and $b{\downarrow} =_\alpha b'{\downarrow}$. Conclude by the property *Structural, ES* of $=_\alpha$ for the ES calculus.

- *Rename, ES*: $e =_\alpha e'$ because $e = e''[x{\leftarrow}b]$ and $e' = e''\{x{\leftarrow}y\}[y{\leftarrow}b]$ with $y \notin \mathsf{vars}(e)$. Again two subcases:
  - $x \in V_{\mathsf{cr}}$ or b abstraction: then $e{\downarrow} = e''{\downarrow}\{x{\leftarrow}b{\downarrow}\}$ and $e'{\downarrow} = e''\{x{\leftarrow}y\}{\downarrow}\{y{\leftarrow}b{\downarrow}\}$. By Point 2 $e''\{x{\leftarrow}y\}{\downarrow}\{y{\leftarrow}b{\downarrow}\} = e''{\downarrow}\{x{\leftarrow}y\}\{y{\leftarrow}b{\downarrow}\}$ because $y \notin \mathsf{bv}(e'')$. Since $y \notin \mathsf{vars}(e'')$, by Point 3 $y \notin \mathsf{bv}(e''{\downarrow})$. Therefore $e''{\downarrow}\{x{\leftarrow}y\}\{y{\leftarrow}b{\downarrow}\} = e''{\downarrow}\{x{\leftarrow}b{\downarrow}\}$ and we conclude.
  - Otherwise, $e{\downarrow} = e''{\downarrow}[x{\leftarrow}b{\downarrow}]$ and $e'{\downarrow} = e''\{x{\leftarrow}y\}{\downarrow}[y{\leftarrow}b{\downarrow}]$. As in the point above, $e''\{x{\leftarrow}y\}{\downarrow}[y{\leftarrow}b{\downarrow}] = e''{\downarrow}\{x{\leftarrow}y\}[y{\leftarrow}b{\downarrow}]$ and we conclude by the points "Rename, ES" of $=_\alpha$ for the ES calculus.
- *Rename, abstraction:* $b =_\alpha b'$ because $b = \lambda x.e$ and $b' = \lambda y.(e\{x{\leftarrow}y\})$ for $y \notin \mathsf{vars}(e)$. Then $b{\downarrow} = \lambda x.(e{\downarrow})$ and $b'{\downarrow} = \lambda y.(e\{x{\leftarrow}y\}{\downarrow})$. By Point 2 $e\{x{\leftarrow}y\}{\downarrow} = e{\downarrow}\{x{\leftarrow}y\}$. By Point 3 and Point 4 $y \notin \mathsf{vars}(e{\downarrow})$, and we conclude by the point "Rename, abstraction" of $=_\alpha$ for the ES calculus.
- *Structural, abstraction:* $b =_\alpha b$ because $b = \lambda x.e$ and $b' = \lambda x.e'$ and $e =_\alpha e'$. Use the *i.h.* and conclude by the point "Structural, abstraction" of $=_\alpha$ for the ES calculus.

2) We prove at the same time the corresponding statement for bites, *i.e.* that $b\{x{\leftarrow}y\}{\downarrow} = b{\downarrow}\{x{\leftarrow}y\}$ when $y \notin \mathsf{bv}(b)$. By structural induction on e and b:

- The cases $e = \epsilon$, $b = z$, and $b = zw$ are trivial.
- If $b = \lambda z.e$, then $b{\downarrow}\{x{\leftarrow}y\} = (\lambda z.(e{\downarrow}))\{x{\leftarrow}y\}$. There are two subcases:
  - If $x = z$, then $(\lambda z.e)\{x{\leftarrow}y\} = \lambda z.e$ and $(\lambda z.(e{\downarrow}))\{x{\leftarrow}y\} = \lambda z.(e{\downarrow})$, and we can conclude.
  - If $x \neq z$, then $(\lambda z.e)\{x{\leftarrow}y\} = \lambda z.(e\{x{\leftarrow}y\})$ and $(\lambda z.(e{\downarrow}))\{x{\leftarrow}y\} = \lambda z.(e{\downarrow}\{x{\leftarrow}y\})$ (recall that $y \notin \mathsf{bv}(e)$, hence $y \notin \mathsf{bv}(e{\downarrow})$ by Point 4, and also $y \neq z$). By *i.h.* $e\{x{\leftarrow}y\}{\downarrow} = e{\downarrow}\{x{\leftarrow}y\}$ and we conclude.
- If $e = e'[z{\leftarrow}b]$, then $e{\downarrow}\{x{\leftarrow}y\} = e'[z{\leftarrow}b]{\downarrow}\{x{\leftarrow}y\}$. There are two subcases:
  - If $z \in V_{\mathsf{cr}}$ or b abstraction, $e'[z{\leftarrow}b]{\downarrow}\{x{\leftarrow}y\} = e'{\downarrow}\{z{\leftarrow}b{\downarrow}\}\{x{\leftarrow}y\}$. Note that $y \neq z$ by the hypothesis that $y \notin \mathsf{bv}(e)$. Two subsubcases:
    - If $x = z$, then $e'{\downarrow}\{z{\leftarrow}b{\downarrow}\}\{x{\leftarrow}y\} = e'{\downarrow}\{z{\leftarrow}b{\downarrow}\{x{\leftarrow}y\}\}$. By *i.h.* $b{\downarrow}\{x{\leftarrow}y\} = b\{x{\leftarrow}y\}{\downarrow}$, and we conclude with $e'{\downarrow}\{z{\leftarrow}b{\downarrow}\{x{\leftarrow}y\}\} = e'[z{\leftarrow}b\{x{\leftarrow}y\}]{\downarrow} = e'[z{\leftarrow}b]\{x{\leftarrow}y\}{\downarrow}$.
    - If $x \neq z$, then $e'{\downarrow}\{z{\leftarrow}b{\downarrow}\}\{x{\leftarrow}y\} = e'{\downarrow}\{x{\leftarrow}y\}\{z{\leftarrow}b{\downarrow}\{x{\leftarrow}y\}\}$. By *i.h.* $e'{\downarrow}\{x{\leftarrow}y\} = e'\{x{\leftarrow}y\}{\downarrow}$ and $b{\downarrow}\{x{\leftarrow}y\} = b\{x{\leftarrow}y\}{\downarrow}$, hence $e'{\downarrow}\{x{\leftarrow}y\}\{z{\leftarrow}b{\downarrow}\{x{\leftarrow}y\}\} = e'\{x{\leftarrow}y\}{\downarrow}\{z{\leftarrow}b\{x{\leftarrow}y\}{\downarrow}\} = e'\{x{\leftarrow}y\}[z{\leftarrow}b\{x{\leftarrow}y\}]{\downarrow} = e'[z{\leftarrow}b]\{x{\leftarrow}y\}{\downarrow}$.
  - Otherwise, $e'[z{\leftarrow}b]{\downarrow}\{x{\leftarrow}y\} = e'{\downarrow}[z{\leftarrow}b{\downarrow}]\{x{\leftarrow}y\}$. Note that $y \neq z$ by the hypothesis that $y \notin \mathsf{bv}(e)$. Two subsubcases:
    - If $x = z$, then $e'{\downarrow}[z{\leftarrow}b{\downarrow}]\{x{\leftarrow}y\} = e'{\downarrow}[z{\leftarrow}b{\downarrow}\{x{\leftarrow}y\}]$. By *i.h.* $b{\downarrow}\{x{\leftarrow}y\} = b\{x{\leftarrow}y\}{\downarrow}$, and we conclude with $e'{\downarrow}[z{\leftarrow}b{\downarrow}\{x{\leftarrow}y\}] = e'[z{\leftarrow}b\{x{\leftarrow}y\}]{\downarrow} = e'[z{\leftarrow}b]\{x{\leftarrow}y\}{\downarrow}$.
    - If $x \neq z$, then $e'{\downarrow}[z{\leftarrow}b{\downarrow}]\{x{\leftarrow}y\} = e'{\downarrow}\{x{\leftarrow}y\}[z{\leftarrow}b{\downarrow}\{x{\leftarrow}y\}]$. By *i.h.* $e'{\downarrow}\{x{\leftarrow}y\} = e'\{x{\leftarrow}y\}{\downarrow}$ and $b{\downarrow}\{x{\leftarrow}y\} = b\{x{\leftarrow}y\}{\downarrow}$, hence $e'{\downarrow}\{x{\leftarrow}y\}[z{\leftarrow}b{\downarrow}\{x{\leftarrow}y\}] = e'\{x{\leftarrow}y\}{\downarrow}[z{\leftarrow}b\{x{\leftarrow}y\}{\downarrow}] = e'\{x{\leftarrow}y\}[z{\leftarrow}b\{x{\leftarrow}y\}]{\downarrow} = e'[z{\leftarrow}b]\{x{\leftarrow}y\}{\downarrow}$.

3) We prove at the same time the corresponding statement for bites, *i.e.* that $\mathsf{fv}(b{\downarrow}) \subseteq \mathsf{fv}(b)$ for each well-named b. By induction on the structure of e and b:

- If $e = \epsilon$, then $\mathsf{fv}(e{\downarrow}) = \mathsf{fv}(\star) = \{\star\}$ and $\mathsf{fv}(e) = \{\star\}$.
- If $e = e'[x{\leftarrow}b]$, then there are two subcases:
  - If $x \in V_{\mathsf{cr}}$ or b is an abstraction, then $e{\downarrow} = e'{\downarrow}\{x{\leftarrow}b{\downarrow}\}$. By Lemma E.16.1 $\mathsf{fv}(e'{\downarrow}\{x{\leftarrow}b{\downarrow}\}) \subseteq \mathsf{fv}(e'{\downarrow}) \setminus \{x\} \cup \mathsf{fv}(b{\downarrow})$. By *i.h.* $\mathsf{fv}(e'{\downarrow}) \subseteq \mathsf{fv}(e')$ and $\mathsf{fv}(b{\downarrow}) \subseteq \mathsf{fv}(b)$, and we conclude because $\mathsf{fv}(e) = \mathsf{fv}(e') \setminus \{x\} \cup \mathsf{fv}(b)$.
  - Otherwise $e{\downarrow} = e'{\downarrow}[x{\leftarrow}b{\downarrow}]$, and $\mathsf{fv}(e{\downarrow}) = \mathsf{fv}(e'{\downarrow}) \setminus \{x\} \cup \mathsf{fv}(b{\downarrow})$. By *i.h.* $\mathsf{fv}(e'{\downarrow}) \subseteq \mathsf{fv}(e')$ and $\mathsf{fv}(b{\downarrow}) \subseteq \mathsf{fv}(b)$, and we conclude.
- If $b = x$, then $\mathsf{fv}(x{\downarrow}) = \mathsf{fv}(x) = \{x\}$.
- If $b = xy$, then $\mathsf{fv}(xy{\downarrow}) = \mathsf{fv}(xy) = \{x, y\}$.
- If $b = \lambda x.e$, then $\mathsf{fv}(\lambda x.e{\downarrow}) = \mathsf{fv}(\lambda x.(e{\downarrow})) = \mathsf{fv}(e{\downarrow}) \setminus \{x\}$. By *i.h.* $\mathsf{fv}(e{\downarrow}) \subseteq \mathsf{fv}(e)$, and we conclude.

4) We prove at the same time the corresponding statement for bites, *i.e.* that $\mathsf{bv}(b{\downarrow}) \subseteq \mathsf{bv}(b)$ for each well-named b. By induction on the structure of e and b:

- If $e = \epsilon$, then $\mathsf{bv}(e{\downarrow}) = \mathsf{bv}(\star) = \emptyset$ and $\mathsf{bv}(e) = \emptyset$.
- If $e = e'[x{\leftarrow}b]$, then there are two subcases:
  - If $x \in V_{\mathsf{cr}}$ or b is an abstraction, then $e{\downarrow} = e'{\downarrow}\{x{\leftarrow}b{\downarrow}\}$. By Lemma E.16.2 $\mathsf{bv}(e'{\downarrow}\{x{\leftarrow}b{\downarrow}\}) \subseteq \mathsf{bv}(e'{\downarrow}) \cup \mathsf{bv}(b{\downarrow})$. By *i.h.* $\mathsf{bv}(e'{\downarrow}) \subseteq \mathsf{bv}(e')$ and $\mathsf{bv}(b{\downarrow}) \subseteq \mathsf{bv}(b)$, and we conclude because $\mathsf{bv}(e) = \mathsf{bv}(e') \cup \{x\} \cup \mathsf{bv}(b)$.

- Otherwise $e{\downarrow} = e'{\downarrow}[x{\leftarrow}b{\downarrow}]$, and $\mathsf{bv}(e{\downarrow}) = \mathsf{bv}(e'{\downarrow}) \cup \{x\} \cup \mathsf{bv}(b{\downarrow})$. By *i.h.* $\mathsf{bv}(e'{\downarrow}) \subseteq \mathsf{bv}(e')$ and $\mathsf{bv}(b{\downarrow}) \subseteq \mathsf{bv}(b)$, and we conclude.

- If $b = x$ then $\mathsf{fv}(x{\downarrow}) = \mathsf{bv}(x) = \emptyset$.
- If $b = xy$ then $\mathsf{bv}(xy{\downarrow}) = \mathsf{bv}(xy) = \emptyset$.
- If $b = \lambda x.e$, then $\mathsf{bv}(\lambda x.e{\downarrow}) = \mathsf{bv}(e{\downarrow})\backslash\{x\}$ and $\mathsf{bv}(\lambda x.e) = \mathsf{bv}(e)\backslash\{x\}$. By *i.h.* $\mathsf{bv}(e{\downarrow}) \subseteq \mathsf{bv}(e)$, and we conclude. □

5) We proceed by induction over the structure of e:
   - If $e = \epsilon$, then $e'[x{\leftarrow}b]e{\downarrow} = e'[x{\leftarrow}b]{\downarrow} = e'{\downarrow}\{x{\leftarrow}b{\downarrow}\} = e'{\downarrow}[x{\leftarrow}[\star{\leftarrow}b]{\downarrow}]$.
   - If $e = e''[y{\leftarrow}b']$, then $e'[x{\leftarrow}b]e{\downarrow} = e'[x{\leftarrow}b]e''[y{\leftarrow}b']{\downarrow} = e'[x{\leftarrow}b]e''{\downarrow}\{y{\leftarrow}b'{\downarrow}\}$. By *i.h.* $e'[x{\leftarrow}b]e''{\downarrow} = e'{\downarrow}\{x{\leftarrow}[\star{\leftarrow}b]e''{\downarrow}\}$, and therefore $e'[x{\leftarrow}b]e''{\downarrow}\{y{\leftarrow}b'{\downarrow}\} = e''{\downarrow}\{x{\leftarrow}[\star{\leftarrow}b]e''{\downarrow}\}\{y{\leftarrow}b'{\downarrow}\}$.
     From the hypothesis $\mathsf{fv}(e') \perp \mathsf{dom}(e)$ it follows that $y \notin \mathsf{fv}(e')$, and therefore $y \notin \mathsf{fv}(e'{\downarrow})$ by Point 3. Hence $e''{\downarrow}\{x{\leftarrow}[\star{\leftarrow}b]e''{\downarrow}\}\{y{\leftarrow}b'{\downarrow}\} = e''{\downarrow}\{x{\leftarrow}[\star{\leftarrow}b]e''{\downarrow}\{y{\leftarrow}b'{\downarrow}\}\} = e''{\downarrow}\{x{\leftarrow}[\star{\leftarrow}b]e''[y{\leftarrow}b']{\downarrow}\}$.

The following lemma is an easy consequence of Lemma E.18.5, but we state it separately because it is used later in the proofs about the machine.

Lemma E.19 (Pristine dissociation)**.** *Let $e[x{\leftarrow}b]$ be a pristine and well-named environment. Then there exists an open context O such that for every pristine environment e' satisfying $\mathsf{fv}(e) \perp \mathsf{dom}(e')$, it holds:*

$$(e[x{\leftarrow}b]e'){\downarrow} = O\langle[\star{\leftarrow}b]e'{\downarrow}\rangle.$$

Proof. By pristinity of $e[x{\leftarrow}b]$, there exists an open context O such that $e{\downarrow} = O\langle x\rangle$ and $x \notin \mathsf{vars}(O)$. Let e' be any environment satisfying the hypotheses. By Lemma E.18.5, $e[x{\leftarrow}b]e'{\downarrow} = e{\downarrow}\{x{\leftarrow}[\star{\leftarrow}b]e'{\downarrow}\} = O\langle[\star{\leftarrow}b]e'{\downarrow}\rangle$. □

E. Read-back is inverse to translation

We now want to prove Lemma VII.4 (proved in Lemma E.22), i.e. that the translation of a well-named λ-term t read-backs to t itself.

The proof depends on the following technical lemma that relates the free and bound variables of a translated term to those of the term to be translated.

Lemma E.20 (On the variables of a translated term)**.** *Let t be a λ-term.*

- $\mathsf{fv}(\underline{t}) = \mathsf{fv}(t)$
- $\mathsf{bv}(\underline{t}) \cap V_{\mathsf{calc}} = \mathsf{bv}(t)$
- $\mathsf{dom}(\underline{t}) \subset V_{\mathsf{cr}}$

Moreover, if t is not a variable and $\overline{t} = (y, e)$:

- $\mathsf{fv}(e) = \mathsf{fv}(t)$
- $\mathsf{bv}(e) \cap V_{\mathsf{calc}} = \mathsf{bv}(t)$
- $\mathsf{dom}(e) \subset V_{\mathsf{cr}}$

Proof. The proof proceeds by mutual induction on the structure of t. Here however we provide the proof only for the principal crumbling translation $\underline{(\cdot)}$, as the proof for $\overline{(\cdot)}$ is similar.

- If $t = x$, then $\mathsf{fv}(\underline{t}) = \mathsf{fv}([\star{\leftarrow}x]) = \{x\}$, $\mathsf{bv}(\underline{t}) = \mathsf{bv}([\star{\leftarrow}x]) = \{\star\}$, and $\mathsf{dom}(\underline{t}) = \{\star\}$, and we can conclude.
- If $t = \lambda x.u$, then $\mathsf{fv}(\underline{t}) = \mathsf{fv}([\star{\leftarrow}\lambda x.\underline{u}]) = \mathsf{fv}(\underline{u}) \setminus \{x\}$, $\mathsf{bv}(\underline{t}) = \{\star, var\} \cup \mathsf{bv}(\underline{u})$, and $\mathsf{dom}(\underline{t}) = \{\star\}$. We conclude by using the *i.h.*
- If $t = up$, then then $\overline{t} = [\star{\leftarrow}xy]ee'$ where $(x, e) := \overline{u}$ and $(y, e') := \overline{p}$. By cases on u and p:
  - If u and p are both variables, then $e = e' = \epsilon$, $\underline{t} = [\star{\leftarrow}xy]$ and $\overline{t} = (z, [z{\leftarrow}xy])$. Clearly $\mathsf{fv}(\underline{t}) = \mathsf{fv}([z{\leftarrow}xy]) = \{x, y\}$, $\mathsf{bv}(\underline{t}) = \{\star\}$ and $\mathsf{bv}(e) = \{z\}$, and we can conclude because $\{\star, z\} \subset V_{\mathsf{cr}}$.
  - If u and p are both not variables, then $e, e' \neq \epsilon$ and $x \in \mathsf{dom}(e)$ and $y \in \mathsf{dom}(e')$. By *i.h.*, $\mathsf{fv}(e) = \mathsf{fv}(u)$, $\mathsf{fv}(e') = \mathsf{fv}(p)$, $\mathsf{bv}(e) \cap V_{\mathsf{calc}} = \mathsf{bv}(u)$, $\mathsf{bv}(e') \cap V_{\mathsf{calc}} = \mathsf{bv}(p)$, and $\mathsf{dom}(e) \cup \mathsf{dom}(e') \subset V_{\mathsf{cr}}$.
    * The requirement $\mathsf{dom}(\underline{t}) \subset V_{\mathsf{cr}}$ follows directly from the *i.h.*, since $\mathsf{dom}(\underline{t}) = \{\star\} \cup \mathsf{dom}(e) \cup \mathsf{dom}(e')$ by Lemma E.10.
    * The requirement $\mathsf{bv}(e) \cap V_{\mathsf{calc}} = \mathsf{bv}(t)$ follows easily, since $\mathsf{bv}(e) = \{\star\} \cup \mathsf{bv}(e) \cup \mathsf{bv}(e')$ by Lemma E.10.
    * As for the free variables, note that by Lemma E.10 $\mathsf{fv}([\star{\leftarrow}xy]ee') = (\{x, y\} \setminus \mathsf{dom}(e) \cup \mathsf{fv}(e)) \setminus \mathsf{dom}(e') \cup \mathsf{fv}(e')$. By *i.h.* $(\{x, y\} \setminus \mathsf{dom}(e) \cup \mathsf{fv}(e)) \setminus \mathsf{dom}(e') \cup \mathsf{fv}(e') = (\{x, y\} \setminus \mathsf{dom}(e) \cup \mathsf{fv}(u)) \setminus \mathsf{dom}(e') \cup \mathsf{fv}(p)$, which is simply $\mathsf{fv}(u) \cup \mathsf{fv}(p)$ because $\mathsf{dom}(e) \perp \mathsf{dom}(e')$ since the crumbling variables are chosen as globally fresh during crumbling, and $\mathsf{fv}(u) \perp \mathsf{dom}(e')$ because $\mathsf{dom}(e')$ are fresh crumbling variables for the same reason.
- The cases when only one among u and p is a variable are similar to the two cases proved above. □

We also need the following technical and rather uninteresting lemma on the auxiliary translation.

Lemma E.21 (On the auxiliary translation)**.** *Let t be a λ-term such that $\overline{t} = (x, e)$. Then:*

1) If $e = \epsilon$, then $t = x$ and $x \in V_{\mathsf{calc}}$.
2) If $e \neq \epsilon$, say $e \coloneqq [x \leftarrow b]e'$, then $\underline{t} = [\star \leftarrow b]e'$ and $x \in V_{\mathsf{cr}}$.

Proof. By inspection of the rules defining the crumbling transformation. □

Lemma E.22 (Crumbling properties)**.** *Let t be a well-named λ-term. Then*

See p. 9 Lemma VII.4

1) Inverse*: $\underline{t}{\downarrow} = t$.*
2) Name*: $\underline{t}$ is well-named.*

Proof.

1) We proceed by induction on the structure of t:
   - If $t = x$, then $\underline{t} = [\star \leftarrow x]$, and clearly $[\star \leftarrow x]{\downarrow} = x = t$.
   - If $t = \lambda x.u$, then $\underline{t} = [\star \leftarrow \lambda x.\underline{u}]$. By definition $[\star \leftarrow \lambda x.\underline{u}]{\downarrow} = \lambda x.(\underline{u}{\downarrow})$, and we conclude by using the *i.h.* $\underline{u}{\downarrow} = u$.
   - If $t = up$, then $\underline{t} = [\star \leftarrow xy]ee'$ where $(x, e) = \overline{u}$ and $(y, e') = \overline{p}$. We proceed by cases on p:
     – If $p = y$, then $y \in V_{\mathsf{calc}}$ and $e' = \epsilon$. We proceed by cases on u:
       ∗ If $u = x$, then also $e = \epsilon$. Thus $\underline{t} = [\star \leftarrow xy]$ and $\underline{t}{\downarrow} = [\star \leftarrow xy]{\downarrow} = xy = up = t$.
       ∗ If $u \neq x$, then $e \neq \epsilon$, say $e = [x \leftarrow b]e''$. By Lemma E.21 $\underline{u} = [\star \leftarrow b]e''$ and $x \in V_{\mathsf{cr}}$, and by Lemma E.20 $y \notin \mathsf{dom}(e) \cup \{x\}$. Hence by Lemma E.18.5 $[\star \leftarrow xy]e{\downarrow} = [\star \leftarrow xy]{\downarrow}\{x \leftarrow [\star \leftarrow b]e''{\downarrow}\}$. By *i.h.* $[\star \leftarrow xy]{\downarrow}\{x \leftarrow [\star \leftarrow b]e''{\downarrow}\} = [\star \leftarrow xy]{\downarrow}\{x \leftarrow t\} = ty$.
     – If $p \neq y$, then $e' \neq \epsilon$, say $e' = [x \leftarrow b]e''$. By Lemma E.21 $\underline{p} = [\star \leftarrow b]e''$ and $y \in V_{\mathsf{cr}}$. By Lemma E.18.5 $[\star \leftarrow xy]ee'{\downarrow} = [\star \leftarrow xy]e{\downarrow}\{y \leftarrow [\star \leftarrow b]e''{\downarrow}\}$. By *i.h.* $[\star \leftarrow xy]e{\downarrow}\{y \leftarrow [\star \leftarrow b]e''{\downarrow}\} = [\star \leftarrow xy]e{\downarrow}\{y \leftarrow p\}$. We now proceed by cases on u:
       ∗ If $u = x$, then also $e = \epsilon$ and $x \in V_{\mathsf{calc}}$. Thus $[\star \leftarrow xy]e{\downarrow}\{y \leftarrow p\} = [\star \leftarrow xy]{\downarrow}\{y \leftarrow p\} = xy\{y \leftarrow p\} = xp$.
       ∗ If $u \neq x$, then $e \neq \epsilon$, say $e = [x \leftarrow b]e''$. By Lemma E.21 $\underline{u} = [\star \leftarrow b]e''$ and $x \in V_{\mathsf{cr}}$. Moreover, $x \neq y$ because they are fresh crumbling variables created in distinct branches of the crumbling transformation. Hence by Lemma E.18.5 $[\star \leftarrow xy]e{\downarrow}\{y \leftarrow p\} = [\star \leftarrow xy]{\downarrow}\{x \leftarrow [\star \leftarrow b]e''{\downarrow}\}\{y \leftarrow p\}$. By *i.h.* $[\star \leftarrow xy]{\downarrow}\{x \leftarrow [\star \leftarrow b]e''{\downarrow}\}\{y \leftarrow p\} = [\star \leftarrow xy]{\downarrow}\{x \leftarrow u\}\{y \leftarrow p\} = xy\{x \leftarrow u\}\{y \leftarrow p\}$. We conclude with $xy\{x \leftarrow u\}\{y \leftarrow p\} = up$ because $y \notin \mathsf{fv}(u)$ since $y \in V_{\mathsf{cr}}$.
2) We prove at the same time the corresponding statement for $\overline{(\cdot)}$, *i.e.* that if t is well-named and $\overline{t} = (x, e)$, then e is well-named. By mutual induction on the size of t:
   - If $t = x$ for some variable x, then $\underline{t} = [\star \leftarrow x]$ which is well-named. $\overline{t} = (x, \epsilon)$ and ϵ is clearly well-named.
   - If $t = \lambda x.u$, then $\underline{t} = [\star \leftarrow \lambda x.\underline{u}]$. By *i.h.* $\underline{u}$ is well-named, thus $[\star \leftarrow \lambda x.\underline{u}]$ is well-named as well, if $x \notin \mathsf{bv}(\underline{u})$ (which holds by Lemma E.20 and well-namedness of t). Similarly for $\overline{t} = (z, [z \leftarrow \lambda x.\underline{u}])$, since $z \notin \mathsf{vars}(\lambda x.\underline{u})$ because it is a fresh local variable.
   - If $t = up$, then $\underline{t} = [\star \leftarrow xy]ee'$ where $(x, e) \coloneqq \overline{u}$ and $(y, e') \coloneqq \overline{p}$. By *i.h.* both e and e' are well-named. We apply twice Lemma E.13 to conclude, but we first need to prove the following disjointedness conditions:
     – $\mathsf{bv}([\star \leftarrow xy]) \perp \mathsf{vars}(e)$ follows from the definition of $\perp$ because $\mathsf{bv}([\star \leftarrow xy]) = \{\star\}$.
     – $\mathsf{fv}([\star \leftarrow xy]) \setminus \mathsf{dom}(e) \perp \mathsf{bv}(e)$, that is $\{x, y\} \setminus \mathsf{dom}(e) \perp \mathsf{bv}(e)$. We proceed by cases on u and p:
       ∗ If $u = x$, then $e = \epsilon$ and we conclude because $\mathsf{bv}(e) = \emptyset$.
       ∗ If $u \neq x$ and $p \neq y$, then $x \in \mathsf{dom}(e)$ and $y \in \mathsf{dom}(e')$. Note that $\mathsf{dom}(e) \perp \mathsf{dom}(e')$ by the definition of crumbling, which always generates fresh crumbling variables. Thus $\{x, y\} \setminus \mathsf{dom}(e) = \{y\}$. By Lemma E.20, $\mathsf{bv}(e) \cap V_{\mathsf{calc}} = \mathsf{bv}(u)$. Conclude again by the property of freshness during crumbling.
       ∗ If $u \neq x$ and $p = y$, then $x \in \mathsf{dom}(e)$ and $e' = \epsilon$. In this case $\{x, y\} \setminus \mathsf{dom}(e) = \{y\}$ because $y \notin V_{\mathsf{cr}}$ while $\mathsf{dom}(e) \subset V_{\mathsf{cr}}$ by Lemma E.20. By well-namedness of t, $y \notin \mathsf{bv}(u)$, and conclude by Lemma E.20.
     – $\mathsf{bv}([\star \leftarrow xy]e) \perp \mathsf{vars}(e')$, that is $\mathsf{bv}(e) \perp \mathsf{vars}(e')$, *i.e.* $\mathsf{bv}(e) \perp \mathsf{fv}(e')$ and $\mathsf{bv}(e) \perp \mathsf{bv}(e')$. By Lemma E.20 $\mathsf{fv}(e') = \mathsf{fv}(p)$, $\mathsf{bv}(e) \cap V_{\mathsf{calc}} = \mathsf{bv}(t)$ and $\mathsf{bv}(e') \cap V_{\mathsf{calc}} = \mathsf{bv}(p)$. From the hypothesis that t is well-named, it follows that $\mathsf{fv}(u) \perp \mathsf{bv}(p)$, and that $\mathsf{bv}(u) \perp \mathsf{bv}(p)$. The remaining requirements follow from the definition of crumbling, where crumbling variables are always chosen to be globally fresh.
     – $\mathsf{fv}([\star \leftarrow xy]e) \setminus \mathsf{dom}(e') \perp \mathsf{bv}(e')$. By Lemma E.10, $\mathsf{fv}([\star \leftarrow xy]e) = \{x, y\} \setminus \mathsf{dom}(e) \cup \mathsf{fv}(e)$, and by Lemma E.20 $\{x, y\} \setminus \mathsf{dom}(e) \cup \mathsf{fv}(e) = \{x, y\} \setminus \mathsf{dom}(e) \cup \mathsf{fv}(u)$. First of all, note that $\mathsf{dom}(e') \perp \mathsf{fv}(u)$ because $\mathsf{dom}(e')$ only contains crumbling variables (Lemma E.20); therefore $\mathsf{fv}([\star \leftarrow xy]e) \setminus \mathsf{dom}(e') = \{x, y\} \setminus \mathsf{dom}(e) \setminus \mathsf{dom}(e') \cup \mathsf{fv}(u)$. We proceed by cases on u, p:
       ∗ If $p = y$ then $e' = \epsilon$, and we conclude because $\mathsf{bv}(e') = \emptyset$.

* If $u = x$ and $p \neq y$, then $x \in V_{\mathsf{calc}}$, $e = \epsilon$, and $y \in \mathsf{dom}(e')$. In this case $\{x, y\} \setminus \mathsf{dom}(e) \setminus \mathsf{dom}(e') \cup \mathsf{fv}(u) = \{x\}$. By well-namedness of t, $\{x, y\} \perp \mathsf{bv}(u)$, and we conclude with $\{x, y\} \perp \mathsf{bv}(u)$ by Lemma E.20.
* If $u \neq x$ and $p \neq y$, then $x \in \mathsf{dom}(e)$ and $y \in \mathsf{dom}(e')$. In this case $\{x, y\} \setminus \mathsf{dom}(e) \setminus \mathsf{dom}(e') \cup \mathsf{fv}(u) = \mathsf{fv}(u)$. By well-namedness of t, $\mathsf{fv}(u) \perp \mathsf{bv}(u)$, and we conclude with $\mathsf{fv}(u) \perp \mathsf{bv}(u)$ by Lemma E.20.

As for the case of $\overline{t} = (z, [z{\leftarrow}xy]ee')$ where $(x, e) \coloneqq \overline{u}$ and $(y, e') \coloneqq \overline{p}$, the proof proceeds in a similar way as above.

□

F. On the properties of $\sigma_{\cdot}$ and $L_{\cdot}$.

Here we collect several properties that relate an environment e and its associated σ_e and L_e.

Lemma E.23. $\mathsf{dom}(\sigma_e) \subseteq \mathsf{dom}(e)$

Proof. By structural induction over e:

- Case ϵ: $\mathsf{dom}(\sigma_e) = \mathsf{dom}(Id) = \emptyset = \mathsf{dom}(\epsilon)$.
- Case $e'[x{\leftarrow}b]$: $\sigma_{e'[x\leftarrow b]}$ is either $\sigma_{e'}$ or $\sigma_{e'}\{x{\leftarrow}b{\downarrow}\}$. Composing $\sigma_{e'}$ with $\{x{\leftarrow}b{\downarrow}\}$ can add x to the domain of $\sigma_{e'}$ and remove an arbitrary number of other variables that, after the substitution, can now be mapped to themselves. Therefore we conclude $\mathsf{dom}(\sigma_{e'}\{x{\leftarrow}b{\downarrow}\}) \subseteq \mathsf{dom}(\sigma_{e'}) \cup \{x\} \subseteq_{\textit{i.h.}} \mathsf{dom}(e') \cup \{x\} = \mathsf{dom}(e'[x{\leftarrow}b])$.

□

Lemma E.24. $\sigma_{ee'} = \sigma_e\sigma_{e'}$

Proof. By structural induction over e':

- Case ϵ: $\sigma_e = \sigma_e\, Id = \sigma_e\sigma_\epsilon$.
- Case $e''[x{\leftarrow}b]$: $\sigma_{ee''[x\leftarrow b]} = \sigma_{ee''}\sigma_{[x\leftarrow b]} =_{\textit{i.h.}} \sigma_e\sigma_{e''}\sigma_{[x\leftarrow b]} = \sigma_e\sigma_{e''[x\leftarrow b]}$.

□

In the following statement we commit a little abuse of notation: we write $\{w{\leftarrow}b{\downarrow}\sigma_e\} \cup \sigma_e$ to mean $\{w{\leftarrow}b{\downarrow}\sigma_e\} \cup (\sigma_e \setminus \{w{\leftarrow}\sigma_e(w)\})$, i.e. the total function that maps w to $b{\downarrow}\sigma_e$ and every other variable x to $\sigma_e(x)$.

Lemma E.25 (Left-to-right induced substitutions and substitution contexts)**.**

1) $\sigma_{[w\leftarrow b]e} = \{w{\leftarrow}b{\downarrow}\sigma_e\} \cup \sigma_e$ *if* $b = v$ *or* $w \in V_{\mathsf{cr}}$ *and* $\sigma_{[w\leftarrow b]e} = \sigma_e$ *otherwise*
2) $L_{[w\leftarrow b]e} = L_e$ *if* $b = v$ *or* $w \in V_{\mathsf{cr}}$ *and* $L_{[w\leftarrow b]e} = [w{\leftarrow}b\sigma_e]L_e$ *otherwise*

Proof. We have to prove that:

1) $\sigma_{[w\leftarrow b]e} = \{w{\leftarrow}b{\downarrow}\sigma_e\} \cup \sigma_e$ if $b = v$ or $w \in V_{\mathsf{cr}}$ and $\sigma_{[w\leftarrow b]e} = \sigma_e$ otherwise. By Lemma E.24, $\sigma_{[w\leftarrow b]e} = \sigma_{[w\leftarrow b]}\sigma_e$. The thesis follows from the definition of composition of substitutions and the definition of $\sigma_{[w\leftarrow b]}$.
2) $L_{[w\leftarrow b]e} = L_e$ if $b = v$ or $w \in V_{\mathsf{cr}}$ and $L_{[w\leftarrow b]e} = [w{\leftarrow}b\sigma_e]L_e$ otherwise. We proceed by structural induction on e.
   - Case ϵ. The property holds by definition of induced substitution context, noticing that $\sigma_\epsilon = Id$.
   - Case $e'[x{\leftarrow}b']$.

$$
\begin{array}{ll}
 & L_{[w\leftarrow b]e'[x\leftarrow b']} \\
= & L_{[w\leftarrow b]e'\sigma_{[x\leftarrow b']}L_{[x\leftarrow b']}} \\
=_{\textit{i.h.}} & \begin{cases} L_{e'\sigma_{[x\leftarrow b']}L_{[x\leftarrow b']}} = L_{e'[x\leftarrow b']} & \\ \quad \text{if } b = v \text{ or } w \in V_{\mathsf{cr}} & \\ & \\ [w{\leftarrow}b\sigma_{e'}]L_{e'\sigma_{[x\leftarrow b']}L_{[x\leftarrow b']}} = & \\ \quad [w{\leftarrow}b\sigma_{e'}]L_{e'[x\leftarrow b']} \quad \text{otherwise.} & \end{cases}
\end{array}
$$

□

Lemma E.26. *If e is well-named and $e(x) = v$ then $\sigma_e(x)$ is a value.*

Proof. If $e(x) = v$ then $e = e_1[x{\leftarrow}v]e_2$ and, by Lemma E.24, $\sigma_e(x) = \sigma_{e_1[x\leftarrow v]e_2}(x) = (\sigma_{e_1}\{x{\leftarrow}v{\downarrow}\}\sigma_{e_2})(x)$. Since $e_1[x{\leftarrow}v]e_2$ is well-named, $x \notin \mathsf{bv}(e_1) \supseteq \mathsf{dom}(e_1) \supseteq_{L.E.23} \mathsf{dom}(\sigma_{e_1})$ and thus $(\sigma_{e_1}\{x{\leftarrow}v{\downarrow}\}\sigma_{e_2})(x) = v{\downarrow}\sigma_{e_2}$, which is a value by definition of substitution. □

G. Read-back is modular

Lemma E.27. $(ee')\downarrow = L_{e'}\langle e\downarrow\sigma_{e'}\rangle$.

See p. 9 Lemma VII.6

Proof. By induction on e'. *Base case*: if $e' = \epsilon$ then $L_{e'} = \langle\cdot\rangle$ and $\sigma_{e'}$ is the identity, so that $L_{e'}\langle e\downarrow\sigma_{e'}\rangle = e\downarrow$. *Inductive case*: if $e' = e''[x\leftarrow b]$ there are two cases.

- $b = v$ or $x \in V_{\mathsf{cr}}$ then
$$\begin{array}{rcl}(ee''[x\leftarrow b])\downarrow & = & (ee'')\downarrow\{x\leftarrow b\downarrow\}\\ & =_{\textit{i.h.}} & L_{e''}\langle e\downarrow\sigma_{e''}\rangle\{x\leftarrow b\downarrow\}\\ & = & L_{e''}\{x\leftarrow b\downarrow\}\langle e\downarrow\sigma_{e''}\{x\leftarrow b\downarrow\}\rangle\\ & = & L_{e''[x\leftarrow b]}\langle e\downarrow\sigma_{e''[x\leftarrow b]}\rangle.\end{array}$$
- Otherwise:
$$\begin{array}{rcl}(ee''[x\leftarrow b])\downarrow & = & (ee'')\downarrow[x\leftarrow b\downarrow]\\ & =_{\textit{i.h.}} & L_{e''}\langle e\downarrow\sigma_{e''}\rangle[x\leftarrow b\downarrow]\\ & = & L_{e''[x\leftarrow b]}\langle e\downarrow\sigma_{e''}\rangle\\ & = & L_{e''[x\leftarrow b]}\langle e\downarrow\sigma_{e''[x\leftarrow b]}\rangle.\end{array}$$

□

APPENDIX F
PROOFS OF SECTION VIII (OPEN CRUMBLING MACHINE)

The aim of this section is to provide all the lemmas required to prove the open machine correct. We do not provide the latter proof explicitly. Instead we shall later extend the open machine to a strong machine by adding a new phase and we shall prove that correct. Therefore the proof of correctness of the strong machine shall entail the one for the open machine and — of course — requires all the lemmas provided in this section.

A. Fundamental property of pristine environments

One fundamental property of pristine environments is the fact that the crumbling variables introduced by the crumbling transformation occur at most once in the resulting environment. In order to prove Lemma VIII.3 (proved in Lemma F.3), we first prove the following two auxiliary lemmas:

Lemma F.1 (Pristine free variables)**.** *If e is pristine and well-named, then* $\mathsf{fv}(e) = \mathsf{fv}(e\downarrow)$.

Proof. One direction is proved in Lemma E.18. We now prove the inclusion $\mathsf{fv}(e) \subseteq \mathsf{fv}(e\downarrow)$. We prove the required statement by mutual induction with the corresponding statement for bites (proved below). By induction on the structure of e:

- If $e = \epsilon$ then $\mathsf{fv}(\epsilon) = \mathsf{fv}(\epsilon\downarrow) = \{\star\}$.
- If $e = e'[y\leftarrow b]$ then, by the definition of pristine, both e' and b are pristine, and $e'\downarrow = O'\langle y\rangle$ for an open context O' such that $y \notin \mathsf{vars}(O')$. By *i.h.* $\mathsf{fv}(e') = \mathsf{fv}(e'\downarrow)$ and $\mathsf{fv}(b) = \mathsf{fv}(b\downarrow)$. We conclude by Lemma E.16.1 and Lemma E.17.1.

Now the corresponding statement for bites. We prove that $\mathsf{fv}(b) \subseteq \mathsf{fv}(b\downarrow)$ for every pristine and well-named bite b:

- If b is a variable, then clearly $\mathsf{fv}(b) = \mathsf{fv}(b\downarrow) = \{b\}$.
- If b is an application $b = xy$, then clearly $\mathsf{fv}(b) = \mathsf{fv}(b\downarrow) = \{x, y\}$.
- If b is an abstraction $b = \lambda x.e$, then by *i.h.* $\mathsf{fv}(e) = \mathsf{fv}(e\downarrow)$. We conclude with $\mathsf{fv}(b) = \mathsf{fv}(e) \setminus \{x\} = \mathsf{fv}(e\downarrow) \setminus \{x\} = \mathsf{fv}(b\downarrow)$. □

Lemma F.2. *Let e be a pristine and well-named environment. If $e\downarrow = O\langle x\rangle$ for some open context O such that $x \notin \mathsf{vars}(O)$, then x occurs at most once in e.*

Proof. We prove the required statement by mutual induction with the corresponding statement for bites (proved below). By induction on the structure of e:

- If $e = \epsilon$ then $e\downarrow = \star$, and the only option is $O = \langle\cdot\rangle$ and $x = \star$.
- If $e = e'[y\leftarrow b]$ then, by the definition of pristine, both e' and b are pristine, and $e'\downarrow = O'\langle y\rangle$ for an open context O' such that $y \notin \mathsf{vars}(O')$. By Lemma F.7 $e\downarrow = O'\langle b'\downarrow\rangle$, and therefore we obtain that $O\langle x\rangle = O'\langle b'\downarrow\rangle$. Recall the hypothesis $x \notin \mathsf{vars}(O)$. We proceed by cases:
  - If $x \in \mathsf{vars}(O')$, then $x \notin \mathsf{vars}(b\downarrow)$ and $e'\downarrow = O'\langle y\rangle = O''\langle x\rangle$ for some O'' such that $x \notin \mathsf{vars}(O'')$. By *i.h.*, x occurs at most once in e'. In order to conclude, it suffices to show that $x \notin \mathsf{vars}([y\leftarrow b])$. As for the bound variables, $x \notin \mathsf{bv}([y\leftarrow b])$ holds by well-namedness of e, because x is free in e. As for the free variables, it follows by Lemma F.1.
  - If $x \notin \mathsf{vars}(O')$, then $O = O'\langle O''\rangle$ where $O''\langle x\rangle = b\downarrow$. By *i.h.*, we obtain that x occurs at most once in b. Note that by well-namedness $x \neq y$. From $x \notin \mathsf{vars}(O')$ we obtain $x \notin \mathsf{vars}(O'\langle y\rangle) = \mathsf{vars}(e'\downarrow)$, and we conclude like in the case above by Lemma F.1.

Now the corresponding statement for bites. We prove that for every pristine and well-named bite b, if $b{\downarrow} = O\langle x\rangle$ for some open context O such that $x \notin \mathsf{vars}(O)$, then x occurs at most once in b:

- If b is a variable, then necessarily $b = x$ which clearly occurs once in b.
- If b is an application, then necessarily $b = xy$ ($O = \langle\cdot\rangle y$) or $b = yx$ ($O = y\langle\cdot\rangle$) for $x \neq y$ (because $x \notin \mathsf{vars}(O)$), and therefore x clearly occurs once in b.
- The case when b is an abstraction is not possible, because in this case $b{\downarrow}$ is an abstraction as well, and thus it cannot hold that $b{\downarrow} = O\langle x\rangle$ for some open context O.

□

As a corollary:

Lemma F.3. *If $e[x{\leftarrow}b]$ is pristine and well-named and $x \neq \star$, then x occurs exactly once in e.*

See p. 10 Lemma VIII.3

Proof. Lemma F.2 and the definition of pristine imply that x occurs at most once in e. From the definition of pristine it also follows that $x \in \mathsf{fv}(e{\downarrow})$. To conclude, just note that the unfolding preserves the names of free variables by Lemma E.18 (the only exception is $\star$, which can be syntactically present in the unfolding, but not in the original environment, for example in the case $\epsilon{\downarrow} = \star$). Thus when a free variable different that $\star$ occurs in the unfolding, it must also occur syntactically in the original environment. □

B. Other fundamental properties of pristine environments

Pristine environments are stable under a certain number of operations that are performed on them during a run of the open machine. These properties are used in the proof of correctness of the machine to show the invariant that certain environments of the machine remain pristine during execution, assuming that the property holds for the initial machine state.

In order to prove the essential properties of pristine environments in Lemma F.8, we first introduce some auxiliary properties of open contexts.

The first one is the fact that open contexts are closed under composition:

Lemma F.4 (Composition of open contexts)**.** *Let O and O' be open contexts. Then their composition $O\langle O'\rangle$ is still a open context.*

Proof. By an easy inspection of the grammar of open evaluation contexts. It can be proved formally by induction on O. □

An easy property about the variables of plugged open contexts:

Lemma F.5. *Let O be an open context, and t be a term. Then*

1) $\mathsf{fv}(O) \subseteq \mathsf{fv}(O\langle t\rangle)$.
2) $\mathsf{bv}(O) \subseteq \mathsf{bv}(O\langle t\rangle)$.

Proof. Easy, by induction on the structure of O. □

The following two lemmas prove properties of open contexts under certain substitutions of terms.

Lemma F.6 (Open contexts and substitutions)**.** *Let O be an open context and σ be a substitution. Then:*

1) $O\langle b\rangle\sigma = O\sigma\langle b\sigma\rangle$ *when* $\mathsf{bv}(O) \perp \mathsf{fv}(\sigma) \cup \mathsf{dom}(\sigma)$.
2) $O\sigma$ is a open context.

Proof.

1) By induction over the structure of O:
   - The case $O = \langle\cdot\rangle$ is trivial.
   - If $O = O't$, then $O\langle b\rangle\sigma = (O'\langle b\rangle t)\sigma = O'\langle b\rangle\sigma(t\sigma)$. By *i.h.* $O'\langle b\rangle\sigma = O'\sigma\langle b\sigma\rangle$. Since $O\sigma = (O'\sigma)(t\sigma)$ we can conclude.
   - The case $O = tO'$ is similar to the one above.
   - Case $O = O'[x{\leftarrow}t]$. $\mathsf{bv}(O) \perp \mathsf{fv}(\sigma) \cup \mathsf{dom}(\sigma)$ implies $x \notin \mathsf{fv}(\sigma) \cup \mathsf{dom}(\sigma)$. Therefore $O'[x{\leftarrow}t]\sigma = O'\sigma[x{\leftarrow}t\sigma]$, and we conclude by *i.h.*
   - Case $O = t[x{\leftarrow}O]$. Again, $\mathsf{bv}(O) \perp \mathsf{fv}(\sigma) \cup \mathsf{dom}(\sigma)$ implies $x \notin \mathsf{fv}(\sigma) \cup \mathsf{dom}(\sigma)$. Therefore $t[x{\leftarrow}O']\sigma = t\sigma[x{\leftarrow}O'\sigma]$, and we conclude by *i.h.*
2) Easy, by induction on the structure of O. □

Lemma F.7. *Let t, u be terms and O be an open context such that $x \notin \mathsf{vars}(O)$ and $\mathsf{bv}(O) \perp \mathsf{fv}(u)$. Then $O\langle t\rangle\{x{\leftarrow}u\} = O\langle t\{x{\leftarrow}u\}\rangle$.*

Proof. By induction on the structure of O:

- When $O = \langle\cdot\rangle$, clearly $O\langle t\rangle\{x\leftarrow u\} = t\{x\leftarrow u\} = O\langle t\{x\leftarrow u\}\rangle$.
- When $O = O'p$, $O\langle t\rangle\{x\leftarrow u\} = O'\langle t\rangle\{x\leftarrow u\}p\{x\leftarrow u\}$. Note that $\mathsf{bv}(O') \subseteq \mathsf{bv}(O)$ and $\mathsf{vars}(O') \subseteq \mathsf{vars}(O)$, and therefore by *i.h.* $O'\langle t\rangle\{x\leftarrow u\} = O'\langle t\{x\leftarrow u\}\rangle$. Moreover, from $x \notin \mathsf{vars}(O)$ it follows that $x \notin \mathsf{fv}(p)$, and therefore $p\{x\leftarrow u\} = p$. As a consequence, $O'\langle t\rangle\{x\leftarrow u\}p\{x\leftarrow u\} = O'\langle t\{x\leftarrow u\}\rangle p = O\langle t\{x\leftarrow u\}\rangle$.
- The case $O = pO'$ is similar to the case above.
- When $O = O'[y\leftarrow p]$, $O\langle t\rangle\{x\leftarrow u\} = O'\langle t\rangle[y\leftarrow p]\{x\leftarrow u\}$. Since by hypothesis $x \notin \mathsf{vars}(O)$ and $y \notin \mathsf{fv}(u)$, $O'\langle t\rangle[y\leftarrow p]\{x\leftarrow u\} = O'\langle t\rangle\{x\leftarrow u\}[y\leftarrow p\{x\leftarrow u\}]$. Since $\mathsf{bv}(O') \subseteq \mathsf{bv}(O)$ and $\mathsf{vars}(O') \subseteq \mathsf{vars}(O)$, by *i.h.* $O'\langle t\rangle\{x\leftarrow u\} = O'\langle t\{x\leftarrow u\}\rangle$, and from the hypothesis that $x \notin \mathsf{vars}(O)$ it follows that $x \notin \mathsf{fv}(p)$ and hence $p\{x\leftarrow u\} = p$.
- The case $O = p[y\leftarrow O']$ is similar to the case above. □

We now have all the ingredients to prove the following important properties of every pristine environment:

Lemma F.8 (Pristine properties)**.**

1) Replacement*: if $e[x\leftarrow b]$ is pristine, then $e[x\leftarrow b']$ is pristine for any pristine b'.*
2) Concatenation*: if $e[x\leftarrow b]$ and $[\star\leftarrow b']e'$ are pristine, then $e[x\leftarrow b']e'$ is pristine, under the requirement that $\mathsf{bv}(e) \perp \mathsf{fv}([\star\leftarrow b']e')$ and $\mathsf{fv}(e) \perp \mathsf{dom}(e')$.*
3) α-Conversion*: if e is well-named and pristine and $e =_\alpha e'$ for e' well-named, then e' is pristine as well.*
4) Renaming*: if e is well-named and pristine then $e\{x\leftarrow y\}$ is pristine when $y \notin \mathsf{bv}(e)$.*

Proof.

1) Trivial by the definition of pristine.
2) By induction on e'. If $e' = \epsilon$ then the statement follows directly from Point 1. Let us suppose now that $e' = e''[y\leftarrow b'']$. By the hypothesis that $[\star\leftarrow b']e'$ is pristine, it follows that $y \in V_{\mathsf{cr}}$, that b'' is pristine, and that $[\star\leftarrow b']e''$ is pristine. Therefore by *i.h.* $e[x\leftarrow b']e''$ is pristine. To prove that $e[x\leftarrow b']e''[y\leftarrow b']$ is pristine, it remains to prove that $e[x\leftarrow b']e''\!\downarrow = O\langle y\rangle$ for some open context O such that $y \notin \mathsf{vars}(O)$.
   First of all, by the pristine hypothesis, $e\!\downarrow = O'\langle x\rangle$ and $[\star\leftarrow b']e''\!\downarrow = O''\langle y\rangle$ for some open contexts O' and O'' such that $x \notin \mathsf{vars}(O')$ and $y \notin \mathsf{vars}(O'')$. We then apply Lemma E.18.5 and obtain that $(e[x\leftarrow b']e'')\!\downarrow = e\!\downarrow\{x\leftarrow[\star\leftarrow b']e''\!\downarrow\} = O'\langle x\rangle\{x\leftarrow O''\langle y\rangle\} = O'\langle O''\langle y\rangle\rangle$, where $O := O'\langle O''\rangle$ is an open context because the composition of weak contexts is still a weak context (see Lemma F.4), and $y \notin \mathsf{vars}(O'\langle O''\rangle)$ because $y \notin \mathsf{vars}(O'')$ and $y \notin \mathsf{vars}(O')$.
3) We prove at the same time the corresponding statement for bites, *i.e.* that if b is pristine and well-named and $b =_\alpha b'$ for b' well-named, then b' is pristine. By structural induction on the derivation of respectively $e =_\alpha e'$ or $b =_\alpha b'$:
   - *Reflexivity:* in this case $e = e'$ and $b = b'$, and therefore trivially e' and b' are pristine.
   - *Symmetry:* $e =_\alpha e'$ because $e' =_\alpha e$. Then by *i.h.* we obtain e' pristine, and we conclude because $=_\alpha$ is symmetric. Similarly for bites.
   - *Transitivity:* $e =_\alpha e'$ because $e =_\alpha e''$ and $e'' =_\alpha e'$. Just use the *i.h.* and use symmetry of $=_\alpha$. Similarly for bites.
   - *Structural, ES:* $e =_\alpha e'$ because $e = e''[x\leftarrow b]$, $e' = e'''[x\leftarrow b']$ where $e'' =_\alpha e'''$ and $b =_\alpha b'$. By *i.h.* e''' and b' are pristine, and it remains to show that $e'''\!\downarrow = O\langle x\rangle$ for some open context O such that $x \notin \mathsf{vars}(O)$. Since e is pristine, $e''\!\downarrow = O\langle x\rangle$ for some open context O such that $x \notin \mathsf{vars}(O)$. Conclude by Lemma E.18.1.
   - *Rename, ES*: $e =_\alpha e'$ because $e = e''[x\leftarrow b]$ and $e' = e''\{x\leftarrow y\}[y\leftarrow b]$ with $y \notin \mathsf{vars}(e)$. By *i.h.* e'' and b are pristine, and it remains to show that $y \in V_{\mathsf{cr}}$ and $e''\{x\leftarrow y\}\!\downarrow = O\langle y\rangle$ for some open context O such that $y \notin \mathsf{vars}(O)$. $y \in V_{\mathsf{cr}}$ because α-equality renames crumbling variables to crumbling variables. As for the readback, since e is pristine, $e''\!\downarrow = O'\langle x\rangle$ for some open context O' such that $x \notin \mathsf{vars}(O')$. Let us take $O := O'$. By Lemma E.18.2, $e''\{x\leftarrow y\}\!\downarrow = e''\!\downarrow\{x\leftarrow y\} = O'\langle x\rangle\{x\leftarrow y\}$. Note from $y \notin \mathsf{vars}(e)$, Lemma F.5 and Lemma E.18 follows that $y \notin \mathsf{vars}(O)$. Finally, by Lemma F.7, $O\langle x\rangle\{x\leftarrow y\} = O\langle y\rangle$ (which requires $x \notin \mathsf{vars}(O)$ and $y \notin \mathsf{bv}(O)$), and we conclude.
   - *Rename, abstraction:* $b =_\alpha b'$ because $b = \lambda x.e$ and $b' = \lambda y.(e\{x\leftarrow y\})$ for $y \notin \mathsf{vars}(e)$. Since e is pristine, by Point 4 $e\{x\leftarrow y\}$ is pristine as well, and we conclude.
   - *Structural, abstraction:* $b =_\alpha b$ because $b = \lambda x.e$ and $b' = \lambda x.e'$ and $e =_\alpha e'$. By *i.h.* e' is pristine, and we conclude.
4) We prove the statement by induction on the structure of e, mutually with the following corresponding statement for bites: if b is well-named and pristine then $b\{x\leftarrow y\}$ is pristine when $y \notin \mathsf{bv}(e)$.
   If $e = \epsilon$, then $e\{x\leftarrow y\} = \epsilon$ and it is therefore pristine. If $e = e'[z\leftarrow b]$, there are two subcases:
   - If $x = z$, then $e\{x\leftarrow y\} = e'[z\leftarrow b\{x\leftarrow y\}]$. By *i.h.* $b\{x\leftarrow y\}$ is pristine, and we can conclude by Point 1.
   - If $x \neq z$, then $e\{x\leftarrow y\} = e'\{x\leftarrow y\}[z\leftarrow b\{x\leftarrow y\}]$ because $y \neq z$ by hypothesis. By *i.h.* $e'\{x\leftarrow y\}$ and $b\{x\leftarrow y\}$ are pristine, and in order to conclude it suffices to show that $e'\{x\leftarrow y\}\!\downarrow = O\langle z\rangle$ for some open context O such that $z \notin \mathsf{vars}(O)$. By pristinity of e it follows that $e'\!\downarrow = O'\langle z\rangle$ for some open context O' such that $z \notin \mathsf{vars}(O')$.
     By Lemma E.18.2 $e'\{x\leftarrow y\}\!\downarrow = e'\!\downarrow\{x\leftarrow y\}$ because $y \notin \mathsf{bv}(e')$ and thus also $y \notin \mathsf{bv}(e'\!\downarrow)$ by Lemma E.18.4. This implies that $e'\{x\leftarrow y\}\!\downarrow = O'\langle z\rangle\{x\leftarrow y\}$, which equals $O'\{x\leftarrow y\}\langle z\{x\leftarrow y\}\rangle$ by Lemma F.6.1, and thus $O'\{x\leftarrow y\}\langle z\rangle$.

We take as the required context $O' := O'\{x\leftarrow y\}$, which is open by Lemma F.6.2. It remains to prove that $z \notin \mathsf{vars}(O'\{x\leftarrow y\})$, which follows from the fact that $x \notin \mathsf{vars}(O)$ and by Lemma F.5 and Lemma E.16.

Finally, the statement for bites. If b is a variable or an application then $b\{x\leftarrow y\}$ is clearly pristine; if $b = \lambda z.e'$, there are again two cases:

- If $x = z$, then $b\{x\leftarrow y\} = b$, and we conclude.
- If $x \neq z$, then $b\{x\leftarrow y\} = \lambda z.(e'\{x\leftarrow y\})$ because $y \notin \mathsf{bv}(b)$ *i.e.* $y \neq z$, and we conclude because by *i.h.* $e'\{x\leftarrow y\}$ is pristine.

□

C. Compilation produces a pristine environment

The initial state of an open machine is obtained by compiling a well-named λ-term, obtaining a pristine environment (Lemma F.9). As we will see, that property of being pristine is preserved during a machine execution, but only on the environments on the left of $\triangleleft$.

Lemma F.9 (Properties of the translation)**.** *Let t be a well-named λ-term. Then $\underline{t}$ is pristine.*

Proof. By induction on the size of t:

- If $t = x$, then $\underline{t} = [\star\leftarrow x]$, which is pristine.
- If $t = \lambda x.u$, then $\underline{t} = [\star\leftarrow\lambda x.\underline{u}]$. Since u is also well-named, the *i.h.* provides that $\underline{u}$ is pristine, and we can conclude.
- If $t = up$, then $\underline{t} = [\star\leftarrow xy]ee'$ where $(x, e) = \overline{u}$ and $(y, e') = \overline{p}$.
  We proceed to prove that every sub-environment of $[\star\leftarrow xy]ee'$ is pristine. The thesis then follows easily. We proceed by induction on the length of the chosen sub-environment:
  - Case length equals 1, *i.e.* $[\star\leftarrow xy]$. Clearly pristine.
  - Case $[\star\leftarrow xy]e''[z\leftarrow b]$ where the ES $[z\leftarrow b]$ belongs to e. By *i.h.* $[\star\leftarrow xy]e''$ is pristine. $z \in V_{\mathsf{cr}}$ because $z \in \mathsf{dom}(e)$ and by Lemma E.20. It remains to prove that $[\star\leftarrow xy]e''\downarrow = O\langle z\rangle$ for some open context O such that $z \in \mathsf{vars}(O)$.
    - If $z = x$ then $e'' = \epsilon$ and we conclude with $O := \langle\cdot\rangle y$.
    - If $z \neq x$ then $e'' \neq \epsilon$. In this case, suppose $e'' = [x\leftarrow b']e'''[w\leftarrow b'']$. By inspection of the definition of crumbling and Lemma E.21, the environment $[\star\leftarrow b']e'''[w\leftarrow b'']$ is an initial sub-environment of $\underline{u}$, which by *i.h.* is pristine. Hence $[\star\leftarrow b']e'''[w\leftarrow b'']\downarrow = O'\langle w\rangle$ for some open context O' such that $z \notin \mathsf{vars}(O')$. By Lemma E.18.5, $[\star\leftarrow xy]e''\downarrow = xy\{x\leftarrow O'\langle w\rangle\}$. Conclude by taking $O := O'\langle w\rangle y$.
  - Case $[\star\leftarrow xy]ee''[z\leftarrow b]$ where the ES $[z\leftarrow b]$ belongs to e'. By *i.h.* $[\star\leftarrow xy]ee''$ is pristine. $z \in V_{\mathsf{cr}}$ because $z \in \mathsf{dom}(e')$ and by Lemma E.20. It remains to prove that $[\star\leftarrow xy]ee''\downarrow = O\langle z\rangle$ for some open context O such that $z \in \mathsf{vars}(O)$. First of all, note that $[\star\leftarrow xy]e\downarrow = [\star\leftarrow xy]\downarrow\{x\leftarrow\underline{t}\downarrow\} = [\star\leftarrow xy]\downarrow\{x\leftarrow t\} = uy$ by the discussion on the previous point and by Point 1.
    - If $z = y$ then $e'' = \epsilon$ and we conclude with $O := t\langle\cdot\rangle$.
    - If $z \neq y$ then $e'' \neq \epsilon$. In this case, suppose $e'' = [y\leftarrow b']e'''[w\leftarrow b'']$. By inspection of the definition of crumbling and by Lemma E.21, the environment $[\star\leftarrow b']e'''[w\leftarrow b'']$ is an initial sub-environment of $\underline{p}$, which by *i.h.* is pristine. Hence $[\star\leftarrow b']e'''[w\leftarrow b'']\downarrow = O'\langle w\rangle$ for some open context O' such that $z \notin \mathsf{vars}(O')$. By Lemma E.18.5, $[\star\leftarrow xy]ee''\downarrow = uy\{y\leftarrow O'\langle w\rangle\} = uO'\langle w\rangle$. Conclude by taking $O := uO'\langle w\rangle$. □

APPENDIX G
PROOFS OF SECTION IX (STRONG CRUMBLING MACHINE)

There are no proofs to be done for Sect. IX. However we include here a bunch of technical lemmas on $e_{(\cdot)}$ that will be used in the rest of the paper.

Lemma G.1. *For every K:*

1) $\mathsf{fv}(e_K) \subseteq \mathsf{fv}(K)$
2) $\mathsf{bv}(e_K) \subseteq \mathsf{bv}(K)$
3) *If K is well-named then e_K is well-named*
4) *If $K\langle e\rangle$ is well-named then e is well-named*
5) *If $K\langle e\rangle$ is well-named then K is well-named*

Proof. Easy structural induction on K. □

Lemma G.2. *Let K be a context. Then*

1) $e_{K\langle\langle\cdot\rangle[x\leftarrow b]\rangle} = [x\leftarrow b]e_K$.
2) $e_{K\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle} = e_K$.

Proof. By induction on K.

1) If $K = \langle\cdot\rangle e'$ then $e_{K\langle\langle\cdot\rangle[x\leftarrow b]\rangle} = [x\leftarrow b]e' = [x\leftarrow b]e_K$ and the statement holds. If $K = e_1[x\leftarrow\lambda y.K']e_2$ then $e_{K\langle\langle\cdot\rangle[x\leftarrow b]\rangle} = e_{K'\langle\langle\cdot\rangle[x\leftarrow b]\rangle}e_2$. By *i.h.*, $e_{K'\langle\langle\cdot\rangle[x\leftarrow b]\rangle} = [x\leftarrow e]K'$, and so $e_{K\langle\langle\cdot\rangle[x\leftarrow b]\rangle} = [x\leftarrow b]e_{K'}e_2 = [x\leftarrow b]e_K$.
2) If $K = \langle\cdot\rangle e'$ then $e_{K\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle} = e_{e[x\leftarrow\lambda y.\langle\cdot\rangle]e'} = e_{\langle\cdot\rangle}e' = e_K$ and the statement holds. If $K = e_1[x\leftarrow\lambda y.K']e_2$ then $e_{K\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle} = e_{e_1[x\leftarrow\lambda y.K'\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle]e_2} = e_{K'\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle}e_2 =_{\textit{i.h.}} e_{K'}e_2 = e_{e_1[x\leftarrow\lambda y.K']e_2} = e_K$.

□

a) Variables of plugged machine contexts: We group here a couple of technical lemmas about the variables of plugged machine contexts.

Lemma G.3. $\mathsf{fv}(b) \subseteq \mathsf{fv}(e) \cup \mathsf{bv}(K\langle e[x\leftarrow b]\rangle)$.

Proof. Easy by induction on the structure of e. □

The following is a generalization of Lemma E.10:

Lemma G.4. *For all machine contexts K and environments e:*

1) $\mathsf{fv}(K\langle e\rangle) \subseteq \mathsf{fv}(e) \setminus V \cup \mathsf{fv}(K)$ *where V is a set of variables that depends on K but not on e, such that $V \subseteq \mathsf{bv}(K)$.*
2) $\mathsf{bv}(K\langle e\rangle) = \mathsf{bv}(K) \cup \mathsf{bv}(e)$
3) $\mathsf{vars}(K\langle e\rangle) = \mathsf{vars}(K) \cup \mathsf{vars}(e)$

Proof. Easy by structural induction on K. □

APPENDIX H
PROOFS OF SECTION X (STRONG IMPLEMENTATION THEOREM)

Following the main part of the paper, we first introduce multi-contexts, their properties and multi-step reduction. Lastly, we address the Strong Implementation Theorem Theorem X.4.2 (proved in Thm. H.38) that requires them.

A. Multi-contexts and their properties

We begin studying properties of multi-contexts and proving Lemma X.1 (proved in Lemma H.3).

B. Multi-contexts and multi-steps reduction

We recall here the terminology introduced in the paper and add some more.

Definition H.1 (Kinds of multi context)**.** *A multi context $\mathbb{C}$ is*

- Normal *if $\mathbb{C}\langle f_{\mathsf{s}}\rangle$ is a strong fireball for every strong fireball f_{s};*
- Proper *if it has at least one hole;*
- Fine *if it is strong and proper.*

We state an auxiliary lemma that shows properties of strong and rigid multi-contexts that are required to prove Lemma X.1.

Lemma H.2. *Let $\mathbb{E}$ and $\mathbb{R}$ be respectively a strong and a rigid multi contexts, and let t be a term.*

1) There exists a term u such that $\mathbb{E}\langle t\rangle = u$.
2) There exists a rigid term r such that $\mathbb{R}\langle t\rangle = r$.

Note that the two points imply that if $\mathbb{E}$ and $\mathbb{R}$ have no holes then they are a term and a rigid term respectively.

Proof. By mutual induction on $\mathbb{E}$ and $\mathbb{R}$.

1) Cases of $\mathbb{E}$:
   - $\mathbb{E} = \langle\cdot\rangle$: obvious.
   - $\mathbb{E} = t$: obvious.
   - $\mathbb{E} = \lambda x.\mathbb{E}'$: it follows by the *i.h.*
   - $\mathbb{E} = \mathbb{R}$: by *i.h.* on rigid contexts.
   - $\mathbb{E} = \mathbb{E}'[x\leftarrow\mathbb{R}]$: by *i.h.* there exists a term p and a rigid term r such that $\mathbb{E}'\langle t\rangle = p$ and $\mathbb{R}\langle t\rangle = r$. Therefore, $\mathbb{E}\langle t\rangle = p[x\leftarrow r]$.
2) Cases of $\mathbb{R}$:
   - $\mathbb{R} = y$: obvious.
   - $\mathbb{R} = \mathbb{R}'\mathbb{E}$: by *i.h.* there exists a rigid term r' and a term u such that $\mathbb{R}'\langle t\rangle = r'$ and $\mathbb{E}\langle t\rangle = u$. Therefore, $\mathbb{R}\langle t\rangle = r'u$.
   - $\mathbb{R} = \mathbb{R}'[x\leftarrow\mathbb{R}'']$: by *i.h.* there exist rigid terms r' and r'' such that $\mathbb{R}'\langle t\rangle = r'$ and $\mathbb{R}''\langle t\rangle = r''$. Therefore, $\mathbb{R}\langle t\rangle = r'[x\leftarrow r'']$.

□

We can now proceed proving Lemma X.1 mutually with the corresponding statement for rigid multi contexts:

Lemma H.3 (Multi step)**.** *Let $\mathbb{E}$ and $\mathbb{R}$ be respectively an external and rigid proper multi context with k holes and $\{a_1, \dots, a_n\} \subseteq \{\mathsf{xm}, \mathsf{xe}\}$. If $t \to_{a_1} \cdots \to_{a_n} u$ then*

See p. 12 Lemma X.1

1) *$\mathbb{R}\langle t\rangle\,(\to_{a_1} \cdots \to_{a_n})^k\,\mathbb{R}\langle u\rangle$ where the i-th sequence of steps has the shape $R_i\langle t\rangle \to_{a_1} \cdots \to_{a_n} R_i\langle u\rangle$ for a rigid context R_i, for every $i \in \{1, \dots, k\}$;*
2) *$\mathbb{E}\langle t\rangle\,(\to_{a_1} \cdots \to_{a_n})^k\,\mathbb{E}\langle u\rangle$ where the i-th sequence of steps has the shape $E_i\langle t\rangle \to_{a_1} \cdots \to_{a_n} E_i\langle u\rangle$ for an external context E_i, for every $i \in \{1, \dots, k\}$.*

Proof. Let us lighten the notation by writing $\to_{a_1 \cdots a_n}$ in place of $(\to_{a_1} \cdots \to_{a_n})$. By mutual induction on $\mathbb{E}$ and $\mathbb{R}$:

1) Cases of $\mathbb{R}$:
   - $\mathbb{R} = y$: trivial.
   - $\mathbb{R} = \mathbb{R}'\mathbb{E}$: we have $\mathbb{R}\langle t\rangle = \mathbb{R}'\langle t\rangle\mathbb{E}\langle t\rangle$ where $\mathbb{R}'$ has k_1 holes, $\mathbb{E}$ has k_2 holes, and $k_1 + k_2 = k$. We deal with the case where $k_1 \neq 0 \neq k_2$. If $k_1 = 0$ then by Lemma H.2 $\mathbb{R}' = \mathbb{R}'\langle t\rangle$ is a rigid term and we consider only $\mathbb{E}\langle t\rangle$, and dually if $k_2 = 0$.
     By *i.h.*, $\mathbb{E}\langle t\rangle \to^{k_2}_{a_1 \cdots a_n} \mathbb{E}\langle u\rangle$ where for the i-th sequence of steps there is an external context E'_i such that the step has the shape $E'_i\langle t\rangle \to_{a_1 \cdots a_n} E'_i\langle u\rangle$ for $i \in \{1, \dots, k_2\}$. Then $\mathbb{R}'\langle t\rangle E'_i$ is a rigid context because by Lemma H.2.2 $\mathbb{R}'\langle t\rangle$ is a rigid term. Then
     $$\mathbb{R}'\langle t\rangle\mathbb{E}\langle t\rangle \to^{k_2}_{a_1 \cdots a_n} \mathbb{R}'\langle t\rangle\mathbb{E}\langle u\rangle$$
     By *i.h.*, $\mathbb{R}'\langle t\rangle \to^{k_1}_{a_1 \cdots a_n} \mathbb{R}'\langle u\rangle$ where for the j-th sequence of steps there is a rigid context R'_j such that the step has the shape $R'_j\langle t\rangle \to_{a_1 \cdots a_n} R'_j\langle u\rangle$ for $j \in \{1, \dots, k_1\}$. Then $R'_j\mathbb{E}\langle u\rangle$ is a rigid context for every j, given that by Lemma H.2.1 $\mathbb{E}\langle u\rangle$ is a term. Therefore, we obtain:
     $$\mathbb{R}'\langle t\rangle\mathbb{E}\langle u\rangle \to^{k_1}_{a_1 \cdots a_n} \mathbb{R}'\langle u\rangle\mathbb{E}\langle u\rangle$$
     Summing up,
     $$\mathbb{R}'\langle t\rangle\mathbb{E}\langle t\rangle \to^{k_1+k_2}_{a_1 \cdots a_n} \mathbb{R}'\langle u\rangle\mathbb{E}\langle u\rangle$$
     The $k_1 + k_2$ rigid contexts of the statement are given by $\mathbb{R}'\langle t\rangle E'_i$ with $i \in \{1, \dots, k_2\}$ followed by $R'_j\mathbb{E}\langle u\rangle$ with $j \in \{1, \dots, k_1\}$.
   - $\mathbb{R} = \mathbb{R}'[y{\leftarrow}\mathbb{R}'']$: we have $\mathbb{R}\langle t\rangle = \mathbb{R}'\langle t\rangle[y{\leftarrow}\mathbb{R}''\langle t\rangle]$ where $\mathbb{R}'$ has k_1 holes, $\mathbb{R}''$ has k_2 holes, and $k_1 + k_2 = k$. We deal with the case where $k_1 \neq 0 \neq k_2$. If $k_1 = 0$ then by Lemma H.2 $\mathbb{R}' = \mathbb{R}'\langle t\rangle$ is a rigid term and we consider only $\mathbb{R}''\langle t\rangle$, and dually if $k_2 = 0$.
     By *i.h.*, $\mathbb{R}'\langle t\rangle \to^{k_1}_{a_1 \cdots a_n} \mathbb{R}'\langle u\rangle$ where for the i-th sequence of steps there is a rigid context R'_i such that the step has the shape $R'_i\langle t\rangle \to_{a_1 \cdots a_n} R'_i\langle u\rangle$ for $i \in \{1, \dots, k_1\}$. Then $R'_i[y{\leftarrow}\mathbb{R}''\langle t\rangle]$ is a rigid context for every i because by Lemma H.2.2 $\mathbb{R}''\langle t\rangle$ is a rigid term, and so
     $$\mathbb{R}'\langle t\rangle[y{\leftarrow}\mathbb{R}''\langle t\rangle] \to^{k_1}_{a_1 \cdots a_n} \mathbb{R}'\langle u\rangle[y{\leftarrow}\mathbb{R}''\langle t\rangle]$$
     By *i.h.*, $\mathbb{R}''\langle t\rangle \to^{k_2}_{a_1 \cdots a_n} \mathbb{R}''\langle u\rangle$ where for the j-th sequence of steps there is a rigid context R''_j such that the step has the shape $R''_j\langle t\rangle \to_{a_1 \cdots a_n} R''_j\langle u\rangle$ for $j \in \{1, \dots, k_2\}$. Then $\mathbb{R}'\langle u\rangle[y{\leftarrow}R''_j]$ is a rigid context for every j, given that by Lemma H.2.2 $\mathbb{R}'\langle u\rangle$ is a term. Therefore, we obtain:
     $$\mathbb{R}'\langle u\rangle[y{\leftarrow}\mathbb{R}''\langle t\rangle] \to^{k_2}_{a_1 \cdots a_n} \mathbb{R}'\langle u\rangle[y{\leftarrow}\mathbb{R}''\langle u\rangle]$$
     Summing up,
     $$\mathbb{R}'\langle t\rangle[y{\leftarrow}\mathbb{R}''\langle t\rangle] \to^{k_1+k_2}_{a_1 \cdots a_n} \mathbb{R}'\langle u\rangle[y{\leftarrow}\mathbb{R}''\langle u\rangle]$$
     The $k_1 + k_2$ rigid contexts of the statement are given by $R'_i[y{\leftarrow}\mathbb{R}''\langle t\rangle]$ with $i \in \{1, \dots, k_1\}$ followed by $\mathbb{R}'\langle u\rangle[y{\leftarrow}R''_j]$ with $j \in \{1, \dots, k_2\}$.

2) Cases of $\mathbb{E}$:
   - $\mathbb{E} = \langle\cdot\rangle$: trivial.
   - $\mathbb{E} = p$: trivial.
   - $\mathbb{E} = \lambda y.\mathbb{E}'$: it follows by the *i.h.*
   - $\mathbb{E} = \mathbb{R}$: by *i.h.* on rigid contexts.
   - $\mathbb{E} = \mathbb{E}'[y{\leftarrow}\mathbb{R}]$: we have $\mathbb{E}\langle t\rangle = \mathbb{E}'\langle t\rangle[y{\leftarrow}\mathbb{R}\langle t\rangle]$ where $\mathbb{E}'$ has k_1 holes, $\mathbb{R}$ has k_2 holes, and $k_1 + k_2 = k$. We deal with the case where $k_1 \neq 0 \neq k_2$. If $k_1 = 0$ then by Lemma H.2 $\mathbb{E}' = \mathbb{E}'\langle t\rangle$ is a term and we consider only $\mathbb{R}\langle t\rangle$, and dually if $k_2 = 0$. By *i.h.*, $\mathbb{E}'\langle t\rangle \to^{k_1}_{a_1 \cdots a_n} \mathbb{E}'\langle u\rangle$ where for the i-th sequence of steps there is an external context E'_i

such that the step has the shape $E'_i\langle t\rangle \to_{a_1\cdots a_n} E'_i\langle u\rangle$ for $i \in \{1,\dots,k_1\}$. Then $E'_i[y{\leftarrow}\mathbb{R}\langle t\rangle]$ is an external context for every i because by Lemma H.2.2 $\mathbb{R}\langle t\rangle$ is a rigid term, and so

$$\mathbb{E}'\langle t\rangle[y{\leftarrow}\mathbb{R}\langle t\rangle] \to^{k_1}_{a_1\cdots a_n} \mathbb{E}'\langle u\rangle[y{\leftarrow}\mathbb{R}\langle t\rangle]$$

By *i.h.*, $\mathbb{R}\langle t\rangle \to^{k_2}_{a_1\cdots a_n} \mathbb{R}\langle u\rangle$ where for the j-th sequence of steps there is a rigid context R_j such that the step has the shape $R_j\langle t\rangle \to_{a_1\cdots a_n} R_j\langle u\rangle$ for $j \in \{1,\dots,k_2\}$. Then $\mathbb{E}'\langle u\rangle[y{\leftarrow}R_j]$ is an external context for every j, given that by Lemma H.2.1 $\mathbb{E}\langle u\rangle$ is a term. Therefore, we obtain:

$$\mathbb{E}'\langle u\rangle[y{\leftarrow}\mathbb{R}\langle t\rangle] \to^{k_2}_{a_1\cdots a_n} \mathbb{E}'\langle u\rangle[y{\leftarrow}\mathbb{R}\langle u\rangle]$$

Summing up,

$$\mathbb{E}'\langle t\rangle[y{\leftarrow}\mathbb{R}\langle t\rangle] \to^{k_1+k_2}_{a_1\cdots a_n} \mathbb{E}'\langle u\rangle[y{\leftarrow}\mathbb{R}\langle u\rangle]$$

The k_1+k_2 external contexts of the statement are given by $E'_i[y{\leftarrow}\mathbb{R}\langle t\rangle]$ with $i \in \{1,\dots,k_1\}$ followed by $\mathbb{E}'\langle u\rangle[y{\leftarrow}R_j]$ with $j \in \{1,\dots,k_2\}$.

□

C. Modular read-back

We prove Lemma X.2 as Lemma H.4.3. The proof requires two auxiliary and uninteresting subparts, $Lemma\ X.2.1$ and $Lemma\ X.2.2$.

Lemma H.4 (Modular read back)**.** *For all environments e, e' and machine contexts K:*

See p. 12 Lemma X.2

1) $(\langle\cdot\rangle e')\downarrow = L_{e'}$
2) $(e[x{\leftarrow}\lambda y.K]e')\downarrow = L_{e'}\langle e\downarrow\{x{\leftarrow}\lambda y.K\downarrow\}\sigma_{e'}\rangle$
3) $K\langle e\rangle\downarrow = K\downarrow\langle e\downarrow\sigma_{e_K}\rangle$

Proof.

1) By induction on e'. *Base case*: if $e' = \epsilon$ then $(\langle\cdot\rangle e')\downarrow = \langle\cdot\rangle = L_\epsilon$. *Inductive case*: let $e' = e''[x{\leftarrow}b]$. If $b = v$ or $x \in V_{\mathsf{cr}}$ then $(\langle\cdot\rangle e''[x{\leftarrow}b])\downarrow = (\langle\cdot\rangle e'')\downarrow\{x{\leftarrow}b\downarrow\} =_{i.h.} L_{e''}\{x{\leftarrow}b\downarrow\} = L_{e''[x{\leftarrow}b]}$. Otherwise $(\langle\cdot\rangle e''[x{\leftarrow}b])\downarrow = (\langle\cdot\rangle e'')\downarrow[x{\leftarrow}b] =_{i.h.} L_{e''}[x{\leftarrow}b] = L_{e''[x{\leftarrow}b]}$.
2) By induction on e', as in the previous point.
3) By induction on K. Cases:
   - *Base*: $K = \langle\cdot\rangle e'$. Note that $e' = e_K$. Then $K\langle e\rangle\downarrow = ee'\downarrow =_{L.VII.6} L_{e'}\langle e\downarrow\sigma_{e'}\rangle =_{P.1} K\downarrow\langle e\downarrow\sigma_{e'}\rangle = K\downarrow\langle e\downarrow\sigma_{e_K}\rangle$.
   - *Inductive*: $K = e''[x{\leftarrow}\lambda y.K']e'$. Then

$$\begin{array}{rcl}
K\langle e\rangle\downarrow & = & e''[x{\leftarrow}\lambda y.K'\langle e\rangle]e'\downarrow \\
& =_{P.2} & L_{e'}\langle e''[x{\leftarrow}\lambda y.K'\langle e\rangle]\downarrow\sigma_{e'}\rangle \\
& = & L_{e'}\langle e''\downarrow\{x{\leftarrow}\lambda y.K'\langle e\rangle\downarrow\}\sigma_{e'}\rangle \\
& =_{i.h.} & L_{e'}\langle e''\downarrow\{x{\leftarrow}\lambda y.K'\downarrow\langle e\downarrow\sigma_{e_{K'}}\rangle\}\sigma_{e'}\rangle \\
& = & L_{e'}\langle e''\downarrow\sigma_{e'}\{x{\leftarrow}\lambda y.K'\downarrow\sigma_{e'}\langle e\downarrow\sigma_{e_{K'}}\sigma_{e'}\rangle\}\rangle \\
& = & (L_{e'}\langle e''\downarrow\sigma_{e'}\{x{\leftarrow}\lambda y.K'\downarrow\sigma_{e'}\}\rangle)\langle e\downarrow\sigma_{e_{K'}}\sigma_{e'}\rangle\rangle \\
& = & (L_{e'}\langle e''\downarrow\{x{\leftarrow}\lambda y.K'\downarrow\}\sigma_{e'}\rangle)\langle e\downarrow\sigma_{e_{K'}}\sigma_{e'}\rangle\rangle \\
& =_{P.2} & K\downarrow\langle e\downarrow\sigma_{e_{K'}}\sigma_{e'}\rangle \\
& = & K\downarrow\langle e\downarrow\sigma_{e_K}\rangle
\end{array}$$

□

D. Properties of frames

Here we collect a number of technical lemmas on frames and frames of a context. They are used in the proof of propagation of the invariants since a few invariants are formulated on frames.

Lemma H.5.

1) $F_{K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle} = F_K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle$.
2) $F_{K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle} = F_K$.

Proof. We have to prove:

1) $F_{K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle} = F_K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle$. We proceed by structural induction on K.
   - Case $\langle\cdot\rangle e'$:

$$\begin{array}{rcl}
F_{e[x{\leftarrow}\lambda y.\langle\cdot\rangle]e'} & = & e[x{\leftarrow}\lambda y.\langle\cdot\rangle] \\
& = & F_{\langle\cdot\rangle e'}\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle
\end{array}$$

- Case $e'[z{\leftarrow}\lambda w.K']e''$:

$$\begin{array}{rcl} F_{e'[z\leftarrow\lambda w.K'\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle]e''} & = & e'[z{\leftarrow}\lambda w.F_{K'\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle}] \\ & =_{\textit{i.h.}} & e'[z{\leftarrow}\lambda w.F_{K'}\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle] \\ & = & e'[z{\leftarrow}\lambda w.F_{K'}]\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle \\ & = & F_{e'[z\leftarrow\lambda w.K']e''}\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle \end{array}$$

2) $F_{K\langle\langle\cdot\rangle[x\leftarrow b]\rangle} = F_K$. We proceed by structural induction on K.
   - Case $\langle\cdot\rangle e'$:

$$\begin{array}{rcl} F_{\langle\cdot\rangle[x\leftarrow b]e'} & = & \langle\cdot\rangle \\ & = & F_{\langle\cdot\rangle e'} \end{array}$$

   - Case $e'[z{\leftarrow}\lambda w.K']e''$:

$$\begin{array}{rcl} F_{e'[z\leftarrow\lambda w.K'\langle\langle\cdot\rangle[x\leftarrow b]\rangle]e''} & = & e'[z{\leftarrow}\lambda w.F_{K'\langle\langle\cdot\rangle[x\leftarrow b]\rangle}] \\ & =_{\textit{i.h.}} & e'[z{\leftarrow}\lambda w.F_{K'}] \\ & = & F_{e'[z\leftarrow\lambda w.K']e''} \end{array}$$

□

Lemma H.6.

1) $F\langle e\rangle{\downarrow} = F{\downarrow}\langle e{\downarrow}\rangle$
2) $F\langle F'\rangle{\downarrow} = F{\downarrow}\langle F'{\downarrow}\rangle$

Proof. We show the first point, the proof of the second point is obtained by simply replacing e with a frame. By induction on F. Cases:

- *Base*: $F = \langle\cdot\rangle$. Then $F\langle e\rangle{\downarrow} = e{\downarrow} = \langle\cdot\rangle{\downarrow}\langle e{\downarrow}\rangle$.
- *Inductive*: $F = e'[x{\leftarrow}\lambda y.F']$. Then

$$\begin{array}{rcl} F\langle e\rangle{\downarrow} & = & e'[x{\leftarrow}\lambda y.F'\langle e\rangle]{\downarrow} \\ & = & e''{\downarrow}\{x{\leftarrow}\lambda y.F'\langle e\rangle{\downarrow}\} \\ & =_{\textit{i.h.}} & e''{\downarrow}\{x{\leftarrow}\lambda y.F'{\downarrow}\langle e{\downarrow}\rangle\} \\ & = & (e''{\downarrow}\{x{\leftarrow}\lambda y.F'{\downarrow}\})\langle e{\downarrow}\rangle \\ & = & F{\downarrow}\langle e{\downarrow}\rangle \end{array}$$

□

E. Invariants

The aim of this section is to define enough invariants on the reachable machine states to be able to prove relaxed β-projection Theorem X.4.1 (proved in Thm. H.36): for each reachable state s we need to prove that:

1) If $s \rightsquigarrow_{\beta_v} s'$ then $s{\downarrow}\ (\rightarrow_{\mathsf{xm}}\rightarrow_{\mathsf{xe}})^+\ s'{\downarrow}$.
2) If $s \rightsquigarrow_{\beta_i} s'$ then $s{\downarrow} \rightarrow^+_{\mathsf{xm}} \equiv s'{\downarrow}$.

Let $s = e[x{\leftarrow}yz]{\triangleleft}K$. The proof requires a few intermediate results, the most important are

1) *Open unfolding*: $e[x{\leftarrow}\langle\cdot\rangle]{\downarrow}$ must be an open context, so that $e[x{\leftarrow}yz]{\downarrow}$ is a top-level redex in an open context.
   To guarantee Open unfolding we ask every body in e_K to be pristine for each reachable state s and moreover e to be pristine if $s = e{\triangleleft}K$ and $e \neq \epsilon$. We call this the *Pristine* invariant. The same invariant is required also to prove the open machine correct, where e_K collapses to the the environment on the right of $\triangleleft$.
2) *Proper unfolding*: $K{\downarrow}$ must be proper, otherwise the redex would disappear during the read-back.
   To guarantee Proper unfolding we introduce a notion of *garbage-free state* and we prove every reachable state to be garbage-free. We call this the *Garbage* invariant.
   In order to show Garbage to be invariant, we shall also introduce an additional, technical invariant that we call *Well-crumbling* and that basically says that some properties of pristine environments are propagated also to the evaluated, no longer pristine parts of the state.
   The proof of the Garbage invariant requires the Well-crumbling invariant to hold. The latter in turn requires the Pristine invariant.
3) *Strong unfolding*: $K{\downarrow}$ must be an external multi-context, so that $K\langle e[x{\leftarrow}yz]\rangle{\downarrow}$ is a top-level redex in an open context in a strong, context, i.e. a redex according to the strong strategy.
   To guarantee Strong unfolding we introduce the notion of *good state* and we prove each reachable state to be good. We call this the *Goodness* invariant.
   In order to show Goodness to be an invariant, we shall also introduce the last invariant, called *Well-named*, that asks every reachable state to be well-named.

The four invariants introduced so far, namely Well-Named, Pristine, Well-Crumbled, Garbage, and Good, are sufficient to prove also the other requirements of a relaxed implementation system.

We now define and show the invariance of each of these statements in the following subsections, H-E1–H-E5.

1) Well-named invariant: The Well-named invariant, required to prove the Goodness invariant, is customary in the abstract machine literature since it guarantees the possibility to drop variable names and use the address of variables in memory instead. As a consequence the α-renaming operation $\cdot^{\alpha}$ is implemented just by physically copying the term.

Definition H.7 (Well-named states)**.** *We say that a state $e \bowtie K$ is well-named if $K\langle e\rangle$ is well-named.*

Theorem H.8 (Well-named invariant)**.** *Let $s = e \bowtie K$ be a state reachable from an initial state s_0. Then s is well-named.*

Proof. By induction on the length of the execution $\rho : s_0 \to_{\mathsf{M}}^{*} s$. If ρ is empty then $s = s_0$ and, by definition of initial state, $s_0 = e_0 \triangleleft \langle\cdot\rangle$ for some well-named and pristine environment e_0: then s_0 is well-named because $\langle\cdot\rangle\langle e_0\rangle = e_0$ is well-named by hypothesis.

If ρ is non-empty we look at the last transition $s' \to_{\mathsf{M}} s$, knowing by *i.h.* that the well-named invariant holds for s':

- $e[x \leftarrow y\, z] \triangleleft K \rightsquigarrow_{\beta_v} e([x \leftarrow b]e'\{w \leftarrow z\}) \triangleleft K$ with $(e_K(y))^{\alpha} = \lambda w.([\star \leftarrow b]e')$ and $e_K(z) = v$ for some v.
  We need to prove that $K\langle e([x \leftarrow b]e'\{w \leftarrow z\})\rangle$ is well-named, under the hypothesis that $K\langle e[x \leftarrow y\, z]\rangle$ is well-named.
  - The bound variables of $K\langle e([x \leftarrow b]e'\{w \leftarrow z\})\rangle$ are all distinct: by Lemma G.4, the bound variables in s are the ones bound in s' plus the bound variables of $[x \leftarrow b]e'\{w \leftarrow z\}$ (excluding x, which was already present in s'). The bound variables of $[x \leftarrow b]e'\{w \leftarrow z\}$, however, are all globally fresh due to α-renaming (excluding x) and by Lemma E.17.2.
  - The bound variables of $K\langle e([x \leftarrow b]e'\{w \leftarrow z\})\rangle$ are distinct from its free variables: we prove that $\mathsf{fv}(s) \subseteq \mathsf{fv}(s')$ and $\mathsf{bv}(s) \subseteq \mathsf{bv}(s') \cup W$ (with W a set of globally fresh new variables disjoint from $\mathsf{vars}(s')$), and then conclude using the *i.h.* $\mathsf{fv}(s') \perp \mathsf{bv}(s')$.
    - *Free variables.* By Lemma G.4, $\mathsf{fv}(s') = \mathsf{fv}(e[x \leftarrow yz]) \setminus V \cup \mathsf{fv}(K)$ and $\mathsf{fv}(s) = \mathsf{fv}(e([x \leftarrow b]e'\{w \leftarrow z\})) \setminus V \cup \mathsf{fv}(K)$ for some set of variables V. By Lemma E.12, $\mathsf{fv}([\star \leftarrow b]e') \subseteq \mathsf{fv}(v) \cup \{w\}$ and by Lemma E.17.1 $\mathsf{fv}([\star \leftarrow b]e'\{w \leftarrow z\}) \subseteq \mathsf{fv}(v) \cup \{z\}$. By Lemma E.10, $\mathsf{fv}(e([x \leftarrow b]e'\{w \leftarrow z\})) \subseteq \mathsf{fv}(e) \setminus \mathsf{dom}([x \leftarrow b]e'\{w \leftarrow z\}) \cup \mathsf{fv}([x \leftarrow b]e'\{w \leftarrow z\})$. Because of the α-renaming performed, the variables in $\mathsf{dom}([x \leftarrow b]e'\{w \leftarrow z\}) \setminus \{x\}$ are globally fresh, and therefore $\mathsf{fv}(e) \setminus \mathsf{dom}([x \leftarrow b]e'\{w \leftarrow z\}) \cup \mathsf{fv}([x \leftarrow b]e'\{w \leftarrow z\}) = \mathsf{fv}(e) \setminus \{x\} \cup \mathsf{fv}([x \leftarrow b]e'\{w \leftarrow z\}) \subseteq \mathsf{fv}(e) \setminus \{x\} \cup \mathsf{fv}(v) \cup \{z\}$.
    - *Bound variables.* By Lemma E.10 and Lemma G.4, $\mathsf{bv}(s') \subseteq \mathsf{bv}(e[x \leftarrow yz]) \cup \mathsf{bv}(K) = \mathsf{bv}(e) \cup \mathsf{bv}([x \leftarrow yz]) \cup \mathsf{bv}(K)$ and $\mathsf{bv}(s) = \mathsf{bv}(e([x \leftarrow b]e'\{w \leftarrow z\})) \cup \mathsf{bv}(K) = \mathsf{bv}(e) \cup \mathsf{bv}([x \leftarrow b]e'\{w \leftarrow z\}) \cup \mathsf{bv}(K)$. Because of the α-renaming performed and Lemma E.17.2, $\mathsf{bv}([x \leftarrow b]e'\{w \leftarrow z\}) = \{x\} \cup W$ where W is a set of new, globally fresh variables.

- $e[x \leftarrow y\, z] \triangleleft K \rightsquigarrow_{\beta_i} e[x \leftarrow b]e' \triangleleft K\langle\langle\cdot\rangle[w \leftarrow z]\rangle$ with $(e_K(y))^{\alpha} = \lambda w.([\star \leftarrow b]e')$ and $e_K(z) = i$ for some inert term i.
  We need to prove that $K\langle e([x \leftarrow b]e'[w \leftarrow z])\rangle$ is well-named, under the hypothesis that $K\langle e[x \leftarrow y\, z]\rangle$ is well-named.
  - The bound variables of $K\langle e([x \leftarrow b]e'[w \leftarrow z])\rangle$ are all distinct: by Lemma G.4, the bound variables in s are the ones bound in s' plus the bound variables of $[x \leftarrow b]e'[w \leftarrow z]$ (excluding x, which was already present in s'). The bound variables of $[x \leftarrow b]e'[w \leftarrow z]$, however, are all globally fresh due to α-renaming (excluding x).
  - The bound variables of $K\langle e([x \leftarrow b]e'[w \leftarrow z])\rangle$ are distinct from its free variables: we prove that $\mathsf{fv}(s) \subseteq \mathsf{fv}(s')$ and $\mathsf{bv}(s) \subseteq \mathsf{bv}(s') \cup W$ (with W a set of globally fresh new variables disjoint from $\mathsf{vars}(s')$), and then conclude using the *i.h.* $\mathsf{fv}(s') \perp \mathsf{bv}(s')$.
    - *Free variables.* By Lemma G.4, $\mathsf{fv}(s') = \mathsf{fv}(e[x \leftarrow yz]) \setminus V \cup \mathsf{fv}(K)$ and $\mathsf{fv}(s) = \mathsf{fv}(e([x \leftarrow b]e'[w \leftarrow z])) \setminus V \cup \mathsf{fv}(K)$ for some set of variables V. By Lemma E.12, $\mathsf{fv}([\star \leftarrow b]e') \subseteq \mathsf{fv}(v) \cup \{w\}$ and by definition $\mathsf{fv}([\star \leftarrow b]e'[w \leftarrow z]) \subseteq \mathsf{fv}(v) \cup \{z\}$. By Lemma E.10, $\mathsf{fv}(e[x \leftarrow b]e'[w \leftarrow z]) \subseteq \mathsf{fv}(e) \setminus \mathsf{dom}([x \leftarrow b]e'[w \leftarrow z]) \cup \mathsf{fv}([x \leftarrow b]e'[w \leftarrow z])$. Because of the α-renaming performed, the variables in $\mathsf{dom}([x \leftarrow b]e'[w \leftarrow z]) \setminus \{x\}$ are globally fresh, and therefore $\mathsf{fv}(e) \setminus \mathsf{dom}([x \leftarrow b]e'[w \leftarrow z]) \cup \mathsf{fv}([x \leftarrow b]e'[w \leftarrow z]) = \mathsf{fv}(e) \setminus \{x\} \cup \mathsf{fv}([x \leftarrow b]e'[w \leftarrow z]) \subseteq \mathsf{fv}(e) \setminus \{x\} \cup \mathsf{fv}(v) \cup \{z\}$.
    - *Bound variables.* By Lemma E.10 and Lemma G.4, $\mathsf{bv}(s') \subseteq \mathsf{bv}(e[x \leftarrow yz]) \cup \mathsf{bv}(K) = \mathsf{bv}(e) \cup \mathsf{bv}([x \leftarrow yz]) \cup \mathsf{bv}(K)$ and $\mathsf{bv}(s) = \mathsf{bv}(e([x \leftarrow b]e'[w \leftarrow z])) \cup \mathsf{bv}(K) = \mathsf{bv}(e) \cup \mathsf{bv}([x \leftarrow b]e'[w \leftarrow z]) \cup \mathsf{bv}(K)$. Because of the α-renaming performed and Lemma E.17.2, $\mathsf{bv}([x \leftarrow b]e'[w \leftarrow z]) = \{x\} \cup W$ where W is a set of new, globally fresh variables.

- $e[x \leftarrow y] \triangleleft K \rightsquigarrow_{\mathsf{ren}} e\{x \leftarrow y\} \triangleleft K$ with $x \neq \star$.
  We need to prove that $K\langle e\{x \leftarrow y\}\rangle$ is well-named, under the hypothesis that $K\langle e[x \leftarrow y]\rangle$ is well-named.
  - The bound variables of $K\langle e\{x \leftarrow y\}\rangle$ are all distinct: by Lemma G.4, the bound variables in s are the ones bound in K plus the ones in $e\{x \leftarrow y\}$. Note that $y \notin \mathsf{vars}(e)$ by well-namedness of s', and hence by Lemma E.17.2 the bound variables of $e\{x \leftarrow y\}$ are the same of e. Therefore we can conclude by using the hypothesis that s' is well-named.
  - The bound variables of $K\langle e\{x \leftarrow y\}\rangle$ are distinct from its free variables: we prove that $\mathsf{fv}(s) \subseteq \mathsf{fv}(s')$ and $\mathsf{bv}(s) \subseteq \mathsf{bv}(s')$, and then conclude using the *i.h.* $\mathsf{fv}(s') \perp \mathsf{bv}(s')$.

* *Free variables.* By Lemma G.4 and Lemma E.17.1, $\mathsf{fv}(K\langle e\{x\leftarrow y\}\rangle) = \mathsf{fv}(e\{x\leftarrow y\}) \setminus V \cup \mathsf{fv}(K) \subseteq \mathsf{fv}(e) \setminus \{x\} \cup \{y\} \setminus V \cup \mathsf{fv}(K)$ and $\mathsf{fv}(K\langle e[x\leftarrow y]\rangle) = \mathsf{fv}(e[x\leftarrow y]) \setminus V \cup \mathsf{fv}(K) = \mathsf{fv}(e) \setminus \{x\} \cup \{y\} \setminus V \cup \mathsf{fv}(K)$ for some $V \subseteq \mathsf{bv}(K)$, and we conclude.
* *Bound variables.* By Lemma G.4 and Lemma E.17.2, $\mathsf{bv}(K\langle e\{x\leftarrow y\}\rangle) = \mathsf{bv}(e\{x\leftarrow y\}) \cup \mathsf{bv}(K) \subseteq \mathsf{bv}(e) \cup \mathsf{bv}(K)$ and $\mathsf{bv}(K\langle e[x\leftarrow y]\rangle) = \mathsf{bv}(e[x\leftarrow y]) \cup \mathsf{bv}(K) = \mathsf{bv}(e) \cup \{y\} \cup \mathsf{fv}(K)$, and we conclude.

- $e[x\leftarrow b]\triangleleft K \rightsquigarrow_{\mathsf{sea}_1} e\triangleleft K\langle\langle\cdot\rangle[x\leftarrow b]\rangle$ when b is an abstraction or when b is y or yz but y is not defined in e_K or $e_K(y)$ is not a value.
  $e\triangleleft K\langle\langle\cdot\rangle[x\leftarrow b]\rangle$ is obviously well-named because plugging the crumbled environment in the machine context has the same result in s' and s.

- $\epsilon\triangleleft K \rightsquigarrow_{\mathsf{sea}_2} \epsilon\triangleright K$
  $e[x\leftarrow b]\triangleright K$ is obviously well-named because plugging the crumbled environment in the machine context has the same result in s' and s.

- $e\triangleright K\langle\langle\cdot\rangle[x\leftarrow b]\rangle \rightsquigarrow_{\mathsf{sea}_3} e[x\leftarrow b]\triangleright K$ where b is a variable or an application.
  $\epsilon\triangleright K$ is obviously well-named because plugging the crumbled environment in the machine context has the same result in s' and s.

- $e\triangleright K\langle\langle\cdot\rangle[x\leftarrow v]\rangle \rightsquigarrow_{\mathsf{gc}} e\triangleright K$ with $x \notin \mathsf{fv}(e)$.
  We need to prove that $K\langle e\rangle$ is well-named, under the hypothesis that $K\langle e[x\leftarrow v]\rangle$ is well-named.
  - The bound variables of s are all distinct: clearly s contains fewer occurrences of bound variables than s'. In fact, by Lemma E.10 and Lemma G.4, the bound variables in s' are the ones bound in s plus x and the bound variables in b. Therefore all the bound variables in s are distinct because all bound variables of s' are distinct by well-namedness.
  - The bound variables of $K\langle e\rangle$ are distinct from its free variables: first of all, by Lemma G.4, $\mathsf{fv}(s') = \mathsf{fv}(e[x\leftarrow v]) \setminus V \cup \mathsf{fv}(K)$ and $\mathsf{fv}(s) = \mathsf{fv}(e) \setminus V \cup \mathsf{fv}(K)$ for some set of variables V. Moreover, from the discussion on the point above, $\mathsf{bv}(s) \subseteq \mathsf{bv}(s')$. Now, note that $\mathsf{fv}(e[x\leftarrow v]) = \mathsf{fv}(e) \setminus \{x\} \cup \mathsf{fv}(b) = \mathsf{fv}(e) \cup \mathsf{fv}(b) \supseteq \mathsf{fv}(e)$ because $x \notin \mathsf{fv}(e)$. Therefore also $\mathsf{fv}(s) \subseteq \mathsf{fv}(s')$, and we conclude by the well-named hypothesis $\mathsf{fv}(s') \perp \mathsf{bv}(s')$.

- $e\triangleright K\langle e'[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle \rightsquigarrow_{\mathsf{sea}_4} e'[x\leftarrow\lambda y.e]\triangleright K$.
  $e'[x\leftarrow\lambda y.e]\triangleright K$ is obviously well-named because plugging the crumbled environment in the machine context has the same result in s' and s.

- $e\triangleright K\langle\langle\cdot\rangle[x\leftarrow\lambda y.e']\rangle \rightsquigarrow_{\mathsf{sea}_5} e'\triangleleft K\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle$ with $x \in \mathsf{fv}(e)$.
  $e'\triangleleft K\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle$ is obviously well-named because plugging the crumbled environment in the machine context has the same result in s' and s.

□

2) Pristine invariant: Previously, we defined pristine environments so to characterize the good properties that are enforced by the translation from λ-terms. We extend the notion of pristine environments to machine states as follows, by considering all the unevaluated environments therein contained:

Definition H.9 (Pristine state)**.** *A state $e \bowtie K$ is* pristine *if every body in e_K is pristine, and also if $\bowtie = \triangleleft$ and $e \neq \epsilon$ then e is pristine.*

The fundamental property of a pristine state was basically already stated in Lemma VIII.3 (proved in Lemma F.3). We now lift that result to machine states.

Theorem H.10 (Pristine invariant)**.** *Let $s = e \bowtie K$ be a state reachable from an initial state s_0. Then s is pristine.*

Proof. By induction on the execution $\rho : s_0 \to^*_{\mathsf{M}} s$. If ρ is empty then $s = s_0$ and, by definition of initial state, $s_0 = e_0\triangleleft\langle\cdot\rangle$ for some well-named environment e_0.

- *Every body in $e_{\langle\cdot\rangle}$ is pristine:* obvious since there are none.
- *e_0 is pristine:* by the definition of initial state and Lemma F.9.

If ρ is non-empty we look at the last transition $s' \to_{\mathsf{M}} s$, knowing by *i.h.* that the pristine invariant holds for s':

- $e[x\leftarrow y\,z]\triangleleft K \rightsquigarrow_{\beta_v} e([x\leftarrow b]e'\{w\leftarrow z\})\triangleleft K$ with $(e_K(y))^\alpha = \lambda w.([\star\leftarrow b]e')$ and $e_K(z) = v$ for some v.
  - *Every body in e_K is pristine:* obvious because the property holds by *i.h.*.
  - *$e([x\leftarrow b]e'\{w\leftarrow z\})$ is pristine:* by *i.h.* $e[x\leftarrow y\,z]$ is pristine. Note that $e_K(y)$ is a value, and therefore by the previous point, the body e'' of $e_K(y)$ is pristine. Therefore, $[\star\leftarrow b]e'$, which is an α-renaming of e'', is pristine by Lemma F.8.3. By Lemma F.8.4 also $[\star\leftarrow b]e'\{w\leftarrow z\}$ is pristine (note that $z \notin \mathsf{bv}(e')$ because $[\star\leftarrow b]e'$ is an α-renaming of e'' where all

bound variables are globally fresh). In order to apply Lemma F.8.2 to conclude that $e([x{\leftarrow}b]e'\{w{\leftarrow}z\})$ is pristine, we need to show that $\mathsf{vars}(e) \perp \mathsf{bv}(e'\{w{\leftarrow}z\})$ (which follows from the fact that $[\star{\leftarrow}b]e'$ is an α-renaming of e'' where all bound variables are globally fresh and Lemma E.17.2) and $\mathsf{bv}(e) \perp \mathsf{fv}([\star{\leftarrow}b]e'\{w{\leftarrow}z\})$ ($\mathsf{fv}([\star{\leftarrow}b]e'\{w{\leftarrow}z\}) \subseteq \mathsf{fv}([\star{\leftarrow}b]e') \setminus \{w\} \cup \{z\}$ by Lemma E.17.1, $\mathsf{fv}([\star{\leftarrow}b]e') = \mathsf{fv}(e'')$ by Lemma E.12, which are different than the bound variables in e by well-namedness).

- $e[x{\leftarrow}y\,z]\triangleleft K \rightsquigarrow_{\beta_i} e[x{\leftarrow}b]e'\triangleleft K\langle\langle\cdot\rangle[w{\leftarrow}z]\rangle$ with $(e_K(y))^\alpha = \lambda w.([\star{\leftarrow}b]e')$ and $e_K(z) = i$ for some inert term i.
  - *Every body in* $e_{K\langle\langle\cdot\rangle[w{\leftarrow}z]\rangle} =_{L.G.2.1} [w{\leftarrow}z]e_K$ *is pristine:* obvious because the property holds by *i.h.* for e_K.
  - $e[x{\leftarrow}b]e'$ *is pristine*: by *i.h.*, $e[x{\leftarrow}y\,z]$ is pristine. Note that $e_K(y)$ is a value, and therefore by the previous point, the body e'' of $e_K(y)$ is pristine. Therefore, $[\star{\leftarrow}b]e'$, which is an α-renaming of e'', is pristine by Lemma F.8.3. In order to apply Lemma F.8.2 to conclude that $e[x{\leftarrow}b]e'$ is pristine, we need to show that $\mathsf{vars}(e) \perp \mathsf{bv}(e')$ (which follows from the fact that $[\star{\leftarrow}b]e'$ is an α-renaming of e'' where all bound variables are globally fresh) and $\mathsf{bv}(e) \perp \mathsf{fv}([\star{\leftarrow}b]e')$ ($\mathsf{fv}([\star{\leftarrow}b]e') = \mathsf{fv}(e'')$ by Lemma E.12, which are different than the bound variables in e by well-namedness).

- $e[x{\leftarrow}y]\triangleleft K \rightsquigarrow_{\mathsf{ren}} e\{x{\leftarrow}y\}\triangleleft K$ with $x \neq \star$.
  - *Every body in* e_K *is pristine:* obvious because the property holds by *i.h.*.
  - $e\{x{\leftarrow}y\}$ *is pristine:* by *i.h.*, $e[x{\leftarrow}y]$ is pristine and so e is pristine. Since s is well-named also $e[x{\leftarrow}y]$ is well-named, and thus $y \notin \mathsf{bv}(e)$. Finally, $e\{x{\leftarrow}y\}$ is pristine by Lemma F.8.4.

- $e[x{\leftarrow}b]\triangleleft K \rightsquigarrow_{\mathsf{sea}_1} e\triangleleft K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle$ when b is an abstraction or when b is y or yz but y is not defined in e_K or $e_K(y)$ is not a value.
  - *Every body in* $e_{K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle} =_{L.G.2.1} [x{\leftarrow}b]e_K$ *is pristine:* by *i.h.*, $e[x{\leftarrow}b]\triangleleft K$ is good and therefore $e[x{\leftarrow}b]$ is pristine and thus b is a pristine bite. By induction on the structure of b one can show that every body in b is pristine. By *i.h.*, the property holds for every body of e_K as well, concluding this case.
  - e *is pristine* (if it is not empty): it follows by the fact that $e[x{\leftarrow}b]$ is pristine.

- $\epsilon\triangleleft K \rightsquigarrow_{\mathsf{sea}_2} \epsilon\triangleright K$.
  - *Every body in* e_K *is pristine:* obvious because the property holds by *i.h.*.
  - Obvious because e is pristine.

- $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle \rightsquigarrow_{\mathsf{sea}_3} e[x{\leftarrow}b]\triangleright K$ where b is a variable or an application.
  - *Every body in* e_K *is pristine:* obvious because the property holds by *i.h.* for every body in $e_{K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle} =_{L.G.2.1} [x{\leftarrow}b]e_K$.
  - Nothing to prove because we are in the $\triangleright$ phase.

- $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle \rightsquigarrow_{\mathsf{gc}} e\triangleright K$ with $x \notin \mathsf{fv}(e)$.
  - *Every body in* e_K *is pristine:* obvious because the property holds by *i.h.* for every body in $e_{K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle} =_{L.G.2.1} [x{\leftarrow}v]e_K$.
  - Nothing to prove because we are in the $\triangleright$ phase.

- $e\triangleright K\langle e'[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle \rightsquigarrow_{\mathsf{sea}_4} e'[x{\leftarrow}\lambda y.e]\triangleright K$.
  - *Every body in* e_K *is pristine:* obvious because the property holds by *i.h.* for every body in $e_{K\langle e'[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle} =_{L.G.2.2} e_K$.
  - Nothing to prove because we are in the $\triangleright$ phase.
- $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}\lambda y.e']\rangle \rightsquigarrow_{\mathsf{sea}_5} e'\triangleleft K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle$ with $x \in \mathsf{fv}(e)$.
  - *Every body in* $e_{K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle} =_{L.G.2.2} e_K$ *is pristine:* obvious because the property holds by *i.h.* for every body in $e_{K\langle\langle\cdot\rangle[x{\leftarrow}\lambda y.e']\rangle} =_{L.G.2.1} [x{\leftarrow}\lambda y.e']e_K$.
  - e' *is pristine:* e' is a body of $e_{K\langle\langle\cdot\rangle[x{\leftarrow}\lambda y.e']\rangle} =_{L.G.2.1} [x{\leftarrow}\lambda y.e']e_K$, and is therefore pristine by the previous point. □

3) Well-crumbled invariant: The Well-crumbling invariant is a technical invariant required to prove the Garbage invariant.

Definition H.11 (Well-crumbling)**.** *An environment e is* well-crumbled *iff*

1) for every decomposition $e = e_1[x{\leftarrow}b]e_2$ such that $x \in V_{\mathsf{cr}}$ and b is not an abstraction, $x \in \mathsf{fv}(e_1{\downarrow})$.
2) the property holds recursively for the body of every abstraction that occurs in e

A state $e \bowtie K$ is well-crumbled *if $K\langle e\rangle$ is.*

Lemma H.12.

1) If e is well-crumbled then e^α also is.
2) If ee' is well-crumbled and $x \notin \mathsf{dom}(e')$ then $e\{x{\leftarrow}y\}e'$ is well-crumbled.

Proof. We need to prove that:

1) If e is well-crumbled then e^α also is. Variables in V_{cr} can only be renamed into variables in V_{cr}, renaming turns non-abstractions into non-abstractions and, by Lemma E.12 and Lemma E.18.1, it does not change the set of free variables in the unfolding of an environment.
2) If ee' is well-crumbled and $x \notin \mathsf{dom}(e')$ then $e\{x{\leftarrow}y\}e'$ is well-crumbled. The substitution $\{x{\leftarrow}y\}$ cannot turn a non-abstraction into an abstraction and, by Lemma E.18.2, $\mathsf{fv}(e\{x{\leftarrow}y\}{\downarrow}) = \mathsf{fv}(e{\downarrow}\{x{\leftarrow}y\}) \supseteq \mathsf{fv}(e{\downarrow}) \setminus \{x\}$. Since $x \notin \mathsf{dom}(e')$, the property follows.

□

Lemma H.13. *Every pristine environment is well-crumbled.*

Proof. For every decomposition $e_1[x{\leftarrow}b]e_2$ of a pristine environment it holds that $e_1{\downarrow} = O\langle x\rangle$ for some open context O. Thus $x \in \mathsf{fv}(e_1{\downarrow})$ as required to be well-crumbled. The fact that the same holds recursively for each body of a pristine environment is trivially proved by structural recursion over e. □

Theorem H.14 (Well-crumbled invariant)**.** *Let $s = e \bowtie K$ be a state reachable from an initial state s_0. Then s is well-crumbled.*

Proof. By induction on the execution $\rho : s_0 \to_{\mathsf{M}}^* s$. If ρ is empty then $s = s_0$ and, by definition of initial state, $s_0 = e_0 \triangleleft \langle\cdot\rangle$ for some well-named and pristine environment e_0: s_0 is well-crumbled by definition if e_0 is well-crumbled, which holds by Lemma H.13.

If ρ is non-empty we look at the last transition $s' \to_{\mathsf{M}} s$, knowing by *i.h.* that the well-crumbled invariant holds for s':

- $e[x{\leftarrow}y\,z]\triangleleft K \rightsquigarrow_{\beta_v} e([x{\leftarrow}b]e'\{w{\leftarrow}z\})\triangleleft K$ with $(e_K(y))^\alpha = \lambda w.([\star{\leftarrow}b]e')$ and $e_K(z) = v$ for some v.
  By *i.h.* $e[x{\leftarrow}y\,z]\triangleleft K$ is well-crumbled. Since $e_K(y)$ is an abstraction that occurs in the well-crumbled state $e[x{\leftarrow}y\,z]\triangleleft K$, its body must be well-crumbled and therefore, by Lemma H.12.1, also $[\star{\leftarrow}b]e'$ is well-crumbled and, by Lemma H.12.2, also $[\star{\leftarrow}b]e'\{w{\leftarrow}z\}$ is well-crumbled. The hypothesis over z used for applying Lemma H.12.2 is trivially satisfied because z is a fresh variable that does not occur in K at all. Since x is bound to a non-abstraction, if $x \in V_{\mathsf{cr}}$ then $x \in \mathsf{fv}(e{\downarrow})$. Therefore $e([x{\leftarrow}b]e'\{w{\leftarrow}z\})$ is well-crumbled. We conclude that $e([x{\leftarrow}b]e'\{w{\leftarrow}z\})\triangleleft K$ is well-crumbled by noting that y and z are bound to abstractions.
- $e[x{\leftarrow}y\,z]\triangleleft K \rightsquigarrow_{\beta_i} e[x{\leftarrow}b]e'\triangleleft K\langle\langle\cdot\rangle[w{\leftarrow}z]\rangle$ with $(e_K(y))^\alpha = \lambda w.([\star{\leftarrow}b]e')$ and $e_K(z) = i$ for some inert term i.
  By *i.h.* $e[x{\leftarrow}y\,z]\triangleleft K$ is well-crumbled. Since $e_K(y)$ is an abstraction that occurs in the well-crumbled state $e[x{\leftarrow}y\,z]\triangleleft K$, its body must be well-crumbled and therefore, by Lemma H.12.1, also $[\star{\leftarrow}b]e'$ is well-crumbled and thus $[\star{\leftarrow}b]e'[w{\leftarrow}z]$ is also well-crumbled because $w \in V_{\mathsf{calc}}$. Since x is bound to a non-abstraction, if $x \in V_{\mathsf{cr}}$ then $x \in \mathsf{fv}(e{\downarrow})$. Therefore $e[x{\leftarrow}b]e'[w{\leftarrow}z]$ is well-crumbled. We conclude that $e[x{\leftarrow}b]e'[w{\leftarrow}z]\triangleleft K$ is well-crumbled by noting that y is bound to an abstraction and that z occurs in the unfolding of the environment because $[w{\leftarrow}z]$ does too because $w \in V_{\mathsf{calc}}$.
- $e[x{\leftarrow}y]\triangleleft K \rightsquigarrow_{\mathsf{ren}} e\{x{\leftarrow}y\}\triangleleft K$ with $x \neq \star$.
  Obvious because by *i.h.* $e[x{\leftarrow}y]\triangleleft K$ is well-crumbled and by Lemma H.12.2.
- $e[x{\leftarrow}b]\triangleleft K \rightsquigarrow_{\mathsf{sea}_1} e\triangleleft K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle$ when b is an abstraction or when b is y or yz but y is not defined in e_K or $e_K(y)$ is not a value.
  $e\triangleleft K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle$ is obviously well-crumbled because $e[x{\leftarrow}b]\triangleleft K$ is well-crumbled by *i.h.* and, by definition of well-crumbled context, both the hypothesis and the conclusion require $K\langle e[x{\leftarrow}b]\rangle$ to be well-crumbled.
- $\epsilon\triangleleft K \rightsquigarrow_{\mathsf{sea}_2} \epsilon\triangleright K$
  $\epsilon\triangleright K$ is obviously well-crumbled because $\epsilon\triangleleft K$ is well-crumbled by *i.h.*.
- $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle \rightsquigarrow_{\mathsf{sea}_3} e[x{\leftarrow}b]\triangleright K$ where b is a variable or an application.
  $e[x{\leftarrow}b]\triangleright K$ is obviously well-crumbled because $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle$ is well-crumbled by *i.h.* and, by definition of well-crumbled context, both the hypothesis and the conclusion require $K\langle e[x{\leftarrow}b]\rangle$ to be well-crumbled.
- $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle \rightsquigarrow_{\mathsf{gc}} e\triangleright K$ with $x \notin \mathsf{fv}(e)$.
  $e\triangleright K$ is obviously well-crumbled because by *i.h.* $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle$ is well-crumbled.
- $e\triangleright K\langle e'[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle \rightsquigarrow_{\mathsf{sea}_4} e'[x{\leftarrow}\lambda y.e]\triangleright K$. To prove that $e'[x{\leftarrow}\lambda y.e]\triangleright K$ is well-crumbled, simply note that $e\triangleright K\langle e'[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle$ is well-crumbled by *i.h.* and, by definition of well-crumbled context, both the hypothesis and the conclusion require $K\langle e'[x{\leftarrow}\lambda y.e]\rangle$ to be well-crumbled.
- $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}\lambda y.e']\rangle \rightsquigarrow_{\mathsf{sea}_5} e'\triangleleft K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle$ with $x \in \mathsf{fv}(e)$.
  $e'\triangleleft K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle$ is obviously well-crumbled because $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}\lambda y.e']\rangle$ is well-crumbled by *i.h.* and, by definition of well-crumbled context, both the hypothesis and the conclusion require $K\langle e[x{\leftarrow}\lambda y.e']\rangle$ to be well-crumbled.

□

4) Garbage invariant: The Garbage invariant basically guarantees that the read-back $K\downarrow$ of a machine context K is a proper multi-context. We formulate the invariant directly on the frame of K, since being garbage-free is a property enforced in the already evaluated part of a state.

Definition H.15 (Garbage-free)**.**

- Environments: *an environment e is garbage-free if $y \in \mathsf{fv}(e)$ implies $y \in \mathsf{fv}(e\downarrow)$.*
- Frames: *a frame F is garbage-free if*
  - *$F = \langle\cdot\rangle$, or*
  - *$F = e[x{\leftarrow}\lambda y.F']$ and*
    - *e is garbage-free,*
    - *$x \in \mathsf{fv}(e)$, and*
    - *F' is garbage-free.*
- States: *a state $s = e \bowtie K$ is garbage-free if F_K is garbage-free and when $\bowtie = \triangleright$ then e is garbage-free.*

Garbage-free frames are decomposable:

Lemma H.16 (Garbage-free decomposition)**.** *$F\langle F'\rangle$ is garbage-free iff F and F' are.*

Proof. By structural induction over F.

- Case $\langle\cdot\rangle$. By definition.
- Case $e[x{\leftarrow}\lambda y.F'']$. We need to prove that $e[x{\leftarrow}\lambda y.F'']$ is garbage-free iff $e[x{\leftarrow}\lambda y.F''\langle F'\rangle]$ is. The property follows from the *i.h.* over F' and the definition of garbage-free context.

□

Theorem H.17 (Garbage-free invariant)**.** *Let $s = e \bowtie K$ be a state reachable from an initial state s_0. Then s is garbage-free.*

Proof. By induction on the execution $\rho : s_0 \to_{\mathsf{M}}^* s$. If ρ is empty then $s = s_0$ and, by definition of initial state, $s_0 = e_0 \triangleleft \langle\cdot\rangle$ for some well-named and pristine environment e_0: then s_0 is garbage-free by definition of garbage-free state.

If ρ is non-empty we look at the last transition $s' \to_{\mathsf{M}} s$, knowing by *i.h.* that the garbage-free invariant holds for s':

- $e[x{\leftarrow}y\,z]\triangleleft K \rightsquigarrow_{\beta_v} e([x{\leftarrow}b]e'\{w{\leftarrow}z\})\triangleleft K$ with $(e_K(y))^\alpha = \lambda w.([\star{\leftarrow}b]e')$ and $e_K(z) = v$ for some v.
  $e([x{\leftarrow}b]e'\{w{\leftarrow}z\})\triangleleft K$ is garbage-free iff F_K is garbage-free. The property holds because, by *i.h.*, $e[x{\leftarrow}y\,z]\triangleleft K$ is garbage-free.
- $e[x{\leftarrow}y\,z]\triangleleft K \rightsquigarrow_{\beta_i} e[x{\leftarrow}b]e'\triangleleft K\langle\langle\cdot\rangle[w{\leftarrow}z]\rangle$ with $(e_K(y))^\alpha = \lambda w.([\star{\leftarrow}b]e')$ and $e_K(z) = i$ for some inert term i.
  $e[x{\leftarrow}b]e'\triangleleft K\langle\langle\cdot\rangle[w{\leftarrow}z]\rangle$ is garbage-free iff $F_{K\langle\langle\cdot\rangle[w{\leftarrow}z]\rangle} =_{L.H.5.2} F_K$ is garbage-free. The property holds because, by *i.h.*, $e[x{\leftarrow}y\,z]\triangleleft F_K$ is garbage-free.
- $e[x{\leftarrow}y]\triangleleft K \rightsquigarrow_{\mathsf{ren}} e\{x{\leftarrow}y\}\triangleleft K$ with $x \neq \star$.
  $e\{x{\leftarrow}y\}\triangleleft K$ is garbage-free iff F_K is garbage-free. The property holds because, by *i.h.*, $e[x{\leftarrow}y]\triangleleft K$ is garbage-free.
- $e[x{\leftarrow}b]\triangleleft K \rightsquigarrow_{\mathsf{sea}_1} e\triangleleft K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle$ when b is an abstraction or when b is y or yz but y is not defined in e_K or $e_K(y)$ is not a value.
  $e\triangleleft K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle$ is garbage-free iff $F_{K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle} =_{L.H.5.2} F_K$. The property holds because F_K is garbage free because, by *i.h.*, $e[x{\leftarrow}b]\triangleleft K$ is garbage-free.
- $\epsilon\triangleleft K \rightsquigarrow_{\mathsf{sea}_2} \epsilon\triangleright K$
  $\epsilon\triangleright K$ is garbage-free iff F_K is garbage-free, which holds by *i.h.*, and ϵ is garbage-free, which is obvious by definition of garbage-free environment.
- $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle \rightsquigarrow_{\mathsf{sea}_3} e[x{\leftarrow}b]\triangleright K$ where b is a variable or an application.
  $e[x{\leftarrow}b]\triangleright K$ is garbage-free iff F_K and $e[x{\leftarrow}b]$ are garbage-free. By *i.h.*, $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle$ is garbage-free, i.e. $F_{K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle} =_{L.H.5.2} F_K$ and e are garbage-free. By the well-crumbled invariant, $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle$ is well-crumbled and therefore, because x is bound to a non-abstraction, if $x \in V_{\mathsf{cr}}$ then $x \in \mathsf{fv}(e\downarrow)$. Therefore any variable that occurs free in b also occurs free in $e[x{\leftarrow}b]\downarrow$ that is either $e\downarrow\{x{\leftarrow}b\}$, if $x \in V_{\mathsf{cr}}$, or $e\downarrow[x{\leftarrow}b]$.
- $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle \rightsquigarrow_{\mathsf{gc}} e\triangleright K$ with $x \notin \mathsf{fv}(e)$.
  $e\triangleright K$ is garbage-free iff F_K and e are garbage-free. The property holds because, by *i.h.*, $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle$ is garbage-free, i.e. e and $F_{K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle} =_{L.H.5.2} F_K$ are garbage-free.
- $e\triangleright K\langle e'[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle \rightsquigarrow_{\mathsf{sea}_4} e'[x{\leftarrow}\lambda y.e]\triangleright K$.
  $e'[x{\leftarrow}\lambda y.e]\triangleright K$ is garbage-free iff K and $e'[x{\leftarrow}\lambda y.e]$ are garbage-free. By *i.h.*, $e\triangleright K\langle e'[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle$ is garbage-free, i.e. $F_{K\langle e'[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle} =_{L.H.5.1} F_K\langle e'[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle$ and e are garbage-free. By Lemma H.16, F_K and $F_{e'[x{\leftarrow}\lambda y.\langle\cdot\rangle]}$ are

garbage-free and thus e' is garbage-free and $x \in \mathsf{fv}(e')$ by definition of garbage-free context. In order to conclude that $e'[x{\leftarrow}\lambda y.e]$ is garbage-free we need to show that every variable that occurs free in $e'[x{\leftarrow}\lambda y.e]$ occurs free in $e'[x{\leftarrow}\lambda y.e]{\downarrow} = e'{\downarrow}\{x{\leftarrow}\lambda y.e{\downarrow}\}$. A variable that occurs free in $e'[x{\leftarrow}\lambda y.e]$ occurs free either in e' and thus in $e'{\downarrow}$ because e' is garbage-free, or in e and thus in $e{\downarrow}$ because e is garbage-free. Therefore it occurs free in $e'{\downarrow}\{x{\leftarrow}\lambda y.e{\downarrow}\}$ because $x \in e{\downarrow}$ because $x \in e'$ and e' is garbage-free.

- $e \triangleright K\langle\langle\cdot\rangle[x{\leftarrow}\lambda y.e']\rangle \rightsquigarrow_{\mathsf{sea}_5} e' \triangleleft K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle$ with $x \in \mathsf{fv}(e)$.
  $e' \triangleleft K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle$ is garbage-free iff $F_{K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle}$ is garbage-free. By *i.h.*, $e \triangleright K\langle\langle\cdot\rangle[x{\leftarrow}\lambda y.e']\rangle$ is garbage-free, i.e. $F_{K\langle\langle\cdot\rangle[x{\leftarrow}\lambda y.e']\rangle} =_{L.H.5.2} F_K$ and e are garbage-free. By Lemma H.16, it is sufficient to prove that $e[x{\leftarrow}\lambda y.\langle\cdot\rangle]$ is garbage-free, i.e. that e is garbage-free, which we already proved, and that $x \in \mathsf{fv}(e)$, which holds by hypothesis.

□

5) Good invariant: The good invariant is definitely the most complex one. The fundamental property, which is part of the requirement for a good state $e \bowtie K$, is that $K{\downarrow}$ is a fine context.

However, in order to show that this holds as an invariant for all reachable states, the notion of good state must be strengthened by imposing strict, technical requirements on various fragments of the machine state.

One such requirement is called *compatibility* and it is imposed on the environment that is being evaluated with respect to the substitution σ_{e_K} originated by the enclosing machine context K.

Definition H.18 (Compatibility with a fireball substitution)**.** *Let f_{s} be a strong fireball. We say that f_{s}* is compatible with a (fireball) substitution *σ if whenever a variable x such that $\sigma(x) = v$ occurs free in f_{s} then it does as the argument of an application. Compatibility for other syntactic categories, e.g. external multi contexts, is defined similarly.*

Another fundamental requirement is called *well-framing* and it is imposed on the frame F_K. The frame F_K is the part of the context that has already been strongly evaluated. The induced property is that $F_K{\downarrow}$ is a normal fine multi-context such that applying the substitution σ_{e_K} does not create new redexes in the already computed part.

The well-framed requirement w.r.t. a substitution is a strengthening of that property that requires it to hold hereditarily, since this is necessary in order to propagate it in the proof that all reachable states are good.

Definition H.19 (Well-framed)**.** *A frame F is* well-framed *w.r.t. a substitution σ if, for every decomposition $F = F'\langle F''\rangle$, $F'{\downarrow}$ is a normal fine multi-context compatible with σ.*

The following lemma is a trivial technical property over well-framed frames.

Lemma H.20. *If $F\langle F'\rangle$ is well-framed w.r.t. σ, then F is well-framed w.r.t. σ.*

Proof. Every decomposition $F = F_1\langle F_2\rangle$ induces a decomposition $F\langle F'\rangle = F_1\langle F_2\langle F'\rangle\rangle$. The statement follows by definition of well-framed frame. □

We are ready to define formally the good property:

Definition H.21 (Good stuff)**.** *An environment e' is* open good *if*

- *$\sigma_{e'}$ is a fireball substitution;*
- *$L_{e'}$ is an inert context.*
- *e' has immediate values.*

A context K is good *when*

- *e_K is open good;*
- *F_K is well-framed w.r.t. σ_{e_K};*
- *$K{\downarrow}$ is a fine context.*

A state $e \bowtie K$ is good *if*

- *K is good and*
- *if $\bowtie = \triangleright$ then $e{\downarrow}$ is a strong fireball compatible with σ_{e_K}.*

As already mentioned, the proof of the Good invariant is quite involved. Before proving that all reachable states are good (Thm. H.32) we need a good number of auxiliary results, which we prove in the following paragraphs. Since SCAM transitions can add or remove ES from the machine state, we are going to show that goodness is stable under the addition and removal of ES, under suitable conditions. In order to do that, we need to prove multiple corresponding properties for multi contexts and compatible substitutions.

a) Basic properties of multi contexts: In this paragraph we prove a couple of general properties of multi contexts that are required in the next paragraph.

The first lemma allows to see terms as (non-proper) multi contexts.

Lemma H.22.

1) Every rigid term r is a rigid multi context with no holes.
2) Every inert context is a fine multi context.

Proof. By an easy inspection of the grammar of multi contexts, and by definition. □

The second lemma shows that the plugging of multi contexts amounts to syntactic substitution, in the case when no variable capture can occur:

Lemma H.23 (Substitution and plugging for multi contexts)**.** *Let $\mathbb{E}$ and $\mathbb{R}$ be a strong and a rigid multi contexts such that they do not capture variables in $\mathsf{fv}(\mathbb{E}')$ and with no free occurrences of x. Then*

1) $\mathbb{R}\langle x\rangle\{x\leftarrow\mathbb{E}'\} = \mathbb{R}\langle\mathbb{E}'\rangle$.
2) $\mathbb{E}\langle x\rangle\{x\leftarrow\mathbb{E}'\} = \mathbb{E}\langle\mathbb{E}'\rangle$.

Proof. By mutual induction on $\mathbb{R}$ and $\mathbb{E}$.

1) *Rigid*. Cases:
   - *Variable*, *i.e.* $\mathbb{R} = y$. Then $\mathbb{R}\langle x\rangle\{x\leftarrow\mathbb{E}'\} = y\{x\leftarrow\mathbb{E}'\} = y = \mathbb{R}\langle\mathbb{E}'\rangle$.
   - *Application*, *i.e.* $\mathbb{R} = \mathbb{R}'\mathbb{E}$. Then
   $$\begin{array}{l}\mathbb{R}\langle x\rangle\{x\leftarrow\mathbb{E}'\} = \mathbb{R}'\langle x\rangle\{x\leftarrow\mathbb{E}'\}\mathbb{E}\langle x\rangle\{x\leftarrow\mathbb{E}'\}\\ =_{\textit{i.h.}}\ \mathbb{R}'\langle\mathbb{E}'\rangle\mathbb{E}\langle\mathbb{E}'\rangle = \mathbb{R}\langle\mathbb{E}'\rangle\end{array}$$
   - *Explicit substitution*, *i.e.* $\mathbb{R} = \mathbb{R}'[y\leftarrow\mathbb{R}'']$. Then $\mathbb{R}\langle x\rangle\{x\leftarrow\mathbb{E}'\} = (\mathbb{R}'\langle x\rangle[y\leftarrow\mathbb{R}''\langle x\rangle])\{x\leftarrow\mathbb{E}'\}$. We have that $y \notin \mathsf{fv}(\mathbb{E}')$ because $\mathbb{R}$ does not capture variables in $\mathsf{fv}(\mathbb{E}')$. Therefore, $(\mathbb{R}'\langle x\rangle[y\leftarrow\mathbb{R}''\langle x\rangle])\{x\leftarrow\mathbb{E}'\} = \mathbb{R}'\langle x\rangle\{x\leftarrow\mathbb{E}'\}[y\leftarrow\mathbb{R}''\langle x\rangle\{x\leftarrow\mathbb{E}'\}]$ without having to rename y in $\mathbb{R}'\langle x\rangle[y\leftarrow\mathbb{R}''\langle x\rangle]$. And then one can continue as expected:
   $$\begin{array}{l}\mathbb{R}'\langle x\rangle\{x\leftarrow\mathbb{E}'\}[y\leftarrow\mathbb{R}''\langle x\rangle\{x\leftarrow\mathbb{E}'\}]\\ =_{\textit{i.h.}}\ \mathbb{R}'\langle\mathbb{E}'\rangle[y\leftarrow\mathbb{R}''\langle\mathbb{E}'\rangle] = \mathbb{R}\langle\mathbb{E}'\rangle\end{array}$$
2) *Strong*. Cases:
   - *Empty*, *i.e.* $\mathbb{E} = \langle\cdot\rangle$. Then $\mathbb{E}\langle x\rangle\{x\leftarrow\mathbb{E}'\} = x\{x\leftarrow\mathbb{E}'\} = \mathbb{E}' = \mathbb{E}\langle\mathbb{E}'\rangle$.
   - *Term*, *i.e.* $\mathbb{E} = t$. Remember that $x \notin \mathsf{fv}(\mathbb{E}) = \mathsf{fv}(t)$. Then $\mathbb{E}\langle x\rangle\{x\leftarrow\mathbb{E}'\} = t\{x\leftarrow\mathbb{E}'\} = t = \mathbb{E}\langle\mathbb{E}'\rangle$.
   - *Abstraction*, *i.e.* $\mathbb{E} = \lambda y.\mathbb{E}''$. Then $\mathbb{E}\langle x\rangle\{x\leftarrow\mathbb{E}'\} = (\lambda y.\mathbb{E}''\langle x\rangle)\{x\leftarrow\mathbb{E}'\}$. We have that $y \notin \mathsf{fv}(\mathbb{E}')$ because $\mathbb{E}$ does not capture variables in $\mathsf{fv}(\mathbb{E}')$. Therefore, $(\lambda y.\mathbb{E}''\langle x\rangle)\{x\leftarrow\mathbb{E}'\} = \lambda y.\mathbb{E}''\langle x\rangle\{x\leftarrow\mathbb{E}'\}$ without having to rename y in $\lambda y.\mathbb{E}''\langle x\rangle$. And then one can continue as expected: $\lambda y.\mathbb{E}''\langle x\rangle\{x\leftarrow\mathbb{E}'\} =_{\textit{i.h.}} \lambda y.\mathbb{E}''\langle\mathbb{E}'\rangle = \mathbb{E}\langle\mathbb{E}'\rangle$.
   - *Rigid*, *i.e.* $\mathbb{E} = \mathbb{R}$. By Point 1.
   - *Explicit substitution*, *i.e.* $\mathbb{E} = \mathbb{E}''[x\leftarrow\mathbb{R}]$. Then $\mathbb{E}\langle x\rangle\{x\leftarrow\mathbb{E}'\} = (\mathbb{E}''\langle x\rangle[y\leftarrow\mathbb{R}\langle x\rangle])\{x\leftarrow\mathbb{E}'\}$. We have that $y \notin \mathsf{fv}(\mathbb{E}')$ because $\mathbb{R}$ does not capture variables in $\mathsf{fv}(\mathbb{E}')$. Therefore, $(\mathbb{E}''\langle x\rangle[y\leftarrow\mathbb{R}\langle x\rangle])\{x\leftarrow\mathbb{E}'\} = \mathbb{E}''\langle x\rangle\{x\leftarrow\mathbb{E}'\}[y\leftarrow\mathbb{R}\langle x\rangle\{x\leftarrow\mathbb{E}'\}]$ without having to rename y in $\mathbb{E}''\langle x\rangle[y\leftarrow\mathbb{R}\langle x\rangle]$. And then one can continue as expected:
   $$\begin{array}{l}\mathbb{E}''\langle x\rangle\{x\leftarrow\mathbb{E}'\}[y\leftarrow\mathbb{R}\langle x\rangle\{x\leftarrow\mathbb{E}'\}]\\ =_{\textit{i.h.}}\ \mathbb{E}''\langle\mathbb{E}'\rangle[y\leftarrow\mathbb{R}\langle\mathbb{E}'\rangle] = \mathbb{R}\langle\mathbb{E}'\rangle\end{array}$$

□

b) Multi contexts and compatible substitutions: The following lemma shows that compatibility of a multi context $\mathbb{E}$ with respect to a substitution σ ensures that some nice properties of $\mathbb{E}$ are preserved in $\mathbb{E}\sigma$.

Lemma H.24 (Multi contexts and compatible substitutions)**.** *Let $\mathbb{R}$ be a rigid multi context and $\mathbb{E}$ and a fine multi context both compatible with a fireball substitution σ. Then*

1) $\mathbb{R}\sigma$ is a rigid multi context. Moreover, if $\mathbb{R}$ is proper then $\mathbb{R}\sigma$ is proper.
2) $\mathbb{E}\sigma$ is an external multi context. Moreover, if $\mathbb{E}$ is proper then $\mathbb{E}\sigma$ is proper.

Proof. By mutual induction on $\mathbb{R}$ and $\mathbb{E}$.

1) *Rigid*. Cases:
   - *Variable*, *i.e.* $\mathbb{R} = x$. Since x does not occur as an argument, by compatibility $\mathbb{R}\sigma = x\sigma = \sigma(x)$ is an inert term. By Lemma H.22.1 $\sigma(x)$ can be seen as a rigid multi context.

- *Application*, *i.e.* $\mathbb{R} = \mathbb{R}'\mathbb{E}$. Note that $\mathbb{R}'$ is compatible with σ and that $\mathbb{E}$ is compatible only if $\mathbb{E} \neq x$. By *i.h.*, $\mathbb{R}'\sigma$ is a rigid multi context. If $\mathbb{E} = x$ then $\mathbb{E}\sigma = x\sigma = \sigma(var)$ which is an inert term and thus a rigid multi context by Lemma H.22.1. If $\mathbb{E} \neq x$ then by *i.h.* $\mathbb{E}\sigma$ is an external multi context. Then $\mathbb{R}\sigma = \mathbb{R}'\sigma\mathbb{E}\sigma$ is a rigid multi context.
  If $\mathbb{R}$ is proper then one among $\mathbb{R}'$ and $\mathbb{E}$ is proper, and properness of $\mathbb{R}\sigma$ follows from the *i.h.*
- *Explicit substitution*, *i.e.* $\mathbb{R} = \mathbb{R}'[x \leftarrow \mathbb{R}'']$. Both $\mathbb{R}'$ and $\mathbb{R}''$ are compatible with σ. By *i.h.*, both $\mathbb{R}'\sigma$ and $\mathbb{R}''\sigma$ are rigid multi contexts. Then $\mathbb{R}\sigma = \mathbb{R}'\sigma\mathbb{R}''\sigma$ is a rigid multi context.
  If $\mathbb{R}$ is proper then one among $\mathbb{R}'$ and $\mathbb{R}''$ is proper, and properness of $\mathbb{R}\sigma$ follows from the *i.h.*

2) *Strong*. Cases:
- *Empty*, *i.e.* $\mathbb{E} = \langle\cdot\rangle$. Trivial, because $\langle\cdot\rangle\sigma = \langle\cdot\rangle$.
- *Term*, *i.e.* $\mathbb{E} = t$. Trivial because every term, and in particular $t\sigma$ is an external multi context.
- *Abstraction*, *i.e.* $\mathbb{E} = \lambda y.\mathbb{E}'$. By *i.h.*, $\mathbb{E}'\sigma$ is an external multi context, and so is $\mathbb{E}\sigma = \lambda y.\mathbb{E}'\sigma$. The moreover par follows from the moreover part of the *i.h.*
- *Rigid*, *i.e.* $\mathbb{E} = \mathbb{R}$. It follows from Point 1.
- *Explicit substitution*, *i.e.* $\mathbb{E} = \mathbb{E}'[x \leftarrow \mathbb{R}]$. Both $\mathbb{E}'$ and $\mathbb{R}$ are compatible with σ. By *i.h.*, both $\mathbb{E}'\sigma$ and $\mathbb{R}\sigma$ are external multi contexts. Then $\mathbb{E}\sigma = \mathbb{E}'\sigma\mathbb{R}\sigma$ is an external multi context.
  If $\mathbb{R}$ is proper then one among $\mathbb{R}'$ and $\mathbb{R}''$ is proper, and properness of $\mathbb{R}\sigma$ follows from the *i.h.*

□

The following two lemmas show that compatibility is preserved both by plugging and by composition of rigid and external multi contexts:

Lemma H.25 (Plugging preserves compatibility)**.** *Let $\mathbb{E}$ and $\mathbb{R}$ be a strong and a rigid multi contexts and f_{s} be a strong fireball such that they are all compatible with σ. Then*

1) $\mathbb{R}\langle f_{\mathsf{s}}\rangle$ is compatible with σ.
2) $\mathbb{E}\langle f_{\mathsf{s}}\rangle$ is compatible with σ.

Proof. By mutual induction on $\mathbb{R}$ and $\mathbb{E}$.

1) *Rigid*. Cases:
- *Variable*, *i.e.* $\mathbb{R} = x$. Then $\mathbb{R}\langle f_{\mathsf{s}}\rangle = x = \mathbb{R}$ is compatible with σ.
- *Application*, *i.e.* $\mathbb{R} = \mathbb{R}'\mathbb{E}''$. It follows immediately from the *i.h.* on $\mathbb{R}'$ and $\mathbb{E}''$, apart when $\mathbb{E}'' = x$ and $\sigma(x) = v$. In such a case however the *i.h.* gives compatibility of $\mathbb{R}'$ which is enough to obtain compatibility of $\mathbb{R}$.
- *Explicit substitution*, *i.e.* $\mathbb{R} = \mathbb{R}'[x \leftarrow \mathbb{R}'']$. It follows immediately from the *i.h.*

2) *Strong*. Cases:
- *Empty*, *i.e.* $\mathbb{E} = \langle\cdot\rangle$. Trivial, because $\mathbb{E}\langle f_{\mathsf{s}}\rangle = f_{\mathsf{s}}$ is compatible with σ by hypothesis.
- *Term*, *i.e.* $\mathbb{E} = t$. Then $\mathbb{E}\langle f_{\mathsf{s}}\rangle = t$ which is compatible by hypothesis.
- *Abstraction*, *i.e.* $\mathbb{E} = \lambda x.f_{\mathsf{s}}$. It follows immediately from the *i.h.*
- *Rigid*, *i.e.* $\mathbb{E} = \mathbb{R}$. By Point 1.
- *Explicit substitution*, *i.e.* $\mathbb{E} = f_{\mathsf{s}}[x \leftarrow \mathbb{R}]$. It follows immediately from the *i.h.*

□

Lemma H.26 (Composition of multi contexts)**.** *Let $\mathbb{E}$ and $\mathbb{R}$ be a strong and a rigid multi contexts, and $\mathbb{E}'$ be a further external multi context. Then*

1) $\mathbb{R}\langle\mathbb{E}'\rangle$ is a rigid multi context.
2) $\mathbb{E}\langle\mathbb{E}'\rangle$ is an external multi context.

Moreover, let $\mathbb{C} \in \{\mathbb{E}, \mathbb{R}\}$ and

1) if both $\mathbb{C}$ and $\mathbb{E}'$ are proper (and thus fine), so does $\mathbb{C}\langle\mathbb{E}'\rangle$.
2) if both $\mathbb{C}$ and $\mathbb{E}'$ are compatible with σ, so does $\mathbb{C}\langle\mathbb{E}'\rangle$.
3) if both $\mathbb{C}$ and $\mathbb{E}'$ are normal, so does $\mathbb{C}\langle\mathbb{E}'\rangle$.

Proof. By mutual induction on $\mathbb{R}$ and $\mathbb{E}$.

1) *Rigid*. Cases:
- *Variable*, *i.e.* $\mathbb{R} = x$. Then $\mathbb{R}\langle\mathbb{E}'\rangle = x$ which is a rigid multi context.
- *Application*, *i.e.* $\mathbb{R} = \mathbb{R}'\mathbb{E}''$. It follows immediately from the *i.h.* on $\mathbb{R}'$ and $\mathbb{E}''$, apart for compatibility with σ when $\mathbb{E}'' = x$ and $\sigma(x) = v$. In such a case however the *i.h.* gives compatibility of $\mathbb{R}'$ which is enough to obtain compatibility of $\mathbb{R}$.
- *Explicit substitution*, *i.e.* $\mathbb{R} = \mathbb{R}'[x \leftarrow \mathbb{R}'']$. It follows immediately from the *i.h.*

2) *Strong*. Cases:
   - *Empty*, *i.e.* $\mathbb{E} = \langle\cdot\rangle$. Trivial.
   - *Term*, *i.e.* $\mathbb{E} = t$. Then $\mathbb{E}\langle\mathbb{E}'\rangle = t$ which is an external multi context.
   - *Abstraction*, *i.e.* $\mathbb{E} = \lambda x.\mathbb{E}'$. It follows immediately from the *i.h.*
   - *Rigid*, *i.e.* $\mathbb{E} = \mathbb{R}$. By Point 1.
   - *Explicit substitution*, *i.e.* $\mathbb{E} = \mathbb{E}'[x{\leftarrow}\mathbb{R}]$. It follows immediately from the *i.h.* □

The following auxiliary, technical lemma allows to extract from a term all the occurrences of a free variable, decomposing the term into a multi context that plugs that variable:

Lemma H.27 (Context extraction from fireballs)**.** *Let t be a well-named term such that $x \in \mathsf{fv}(t)$ and t is compatible with $\{x{\leftarrow}v\}$.*

1) if t is a strong inert term then there exists a normal and proper rigid multi context $\mathbb{R}$ such that $t = \mathbb{R}\langle x\rangle$ and $x \notin \mathsf{fv}(\mathbb{R})$.
2) if t is a strong fireball then there exists a fine and normal multi context $\mathbb{E}$ such that $t = \mathbb{E}\langle x\rangle$ and $x \notin \mathsf{fv}(\mathbb{E})$.

Proof. By induction on t. Cases:

- *Variable*, *i.e.* $t = x$:
  1) trivially true, as the compatibility hypothesis is not verified (x occurs free but not as an argument);
  2) Simply take $\mathbb{E} \coloneqq \langle\cdot\rangle$.
- *Application*, *i.e.* $t = i_{\mathsf{s}} f_{\mathsf{s}}$:
  1) Suppose that x occurs in both i_{s} and f_{s}. Note that i_{s} is compatible and that f_{s} is compatible only if $f_{\mathsf{s}} \neq x$. By *i.h.* there is a normal and proper rigid normal multi context $\mathbb{R}'$ such that $i_{\mathsf{s}} = \mathbb{R}'\langle x\rangle$ and $x \notin \mathsf{fv}(\mathbb{R}')$. If $f_{\mathsf{s}} = x$ then take $\mathbb{E} \coloneqq \langle\cdot\rangle$, otherwise by *i.h.* there exists a fine and normal multi context $\mathbb{E}$ such that $f_{\mathsf{s}} = \mathbb{E}\langle x\rangle$ and $x \notin \mathsf{fv}(\mathbb{E})$. Then $\mathbb{R} \coloneqq \mathbb{R}'\mathbb{E}$ satisfies the statement.
     If x does not occur in i_{s} then one uses Lemma H.22.1 to see i_{s} as a normal rigid multi context and reason as before. If x does not occur in f_{s} then one sees f_{s} as a normal and fine multi context with no holes, as all terms are external multi contexts. Note that x has to occur in i or f_{s}, because it occurs in t, and so the context $\mathbb{R}$ is always proper.
  2) Simply take $\mathbb{E}$ as the rigid multi context obtained in the previous point.
- *Abstraction*, *i.e.* $t = \lambda y.f_{\mathsf{s}}$:
  1) trivially true, as the hypothesis is not verified;
  2) Simply take $\mathbb{E} \coloneqq \lambda y.\mathbb{E}'$, where $\mathbb{E}'$ is the fine and normal multi context given by the *i.h.* on f_{s}.
- *Explicit substitution*, *i.e.* $t = u[y{\leftarrow}i_{\mathsf{s}}]$:
  1) if t is a strong inert term then so is u. Suppose that x occurs in both u and i. Note that both u and i_{s} are compatible with $\{x{\leftarrow}v\}$. Therefore, we can apply the *i.h.* on both terms, obtaining two normal and proper rigid multi context $\mathbb{R}'$ and $\mathbb{R}''$ such that $\mathbb{R}'\langle x\rangle = u$, $\mathbb{R}''\langle x\rangle = i_{\mathsf{s}}$, $x \notin \mathsf{fv}(\mathbb{R}')$, and $x \notin \mathsf{fv}(\mathbb{R}'')$. Then $\mathbb{R} \coloneqq \mathbb{R}'[y{\leftarrow}\mathbb{R}'']$ verifies the statement. If x does not occur in one among u and i then one uses Lemma H.22.1 to see it as a normal rigid multi context and reason as before. Note that x has to occur in u or i, because it occurs in t, and so the context $\mathbb{R}$ is always proper.
  2) Along the lines of the previous point, spelled out in the following. If t is a strong fireball then so is u. Suppose that x occurs in both u and i_{s}. We can apply the *i.h.* obtaining a fine and normal multi context $\mathbb{E}'$ such that $f_{\mathsf{s}} = \mathbb{E}\langle x\rangle$ and $x \notin \mathsf{fv}(\mathbb{E})$. By the compatibility hypothesis i_{s} does not have shape $L\langle x\rangle$, and so we can apply the *i.h.*, obtaining a normal and proper rigid multi context $\mathbb{R}'$ such that $\mathbb{R}'\langle x\rangle = i_{\mathsf{s}}$ and $x \notin \mathsf{fv}(\mathbb{R}')$. Then $\mathbb{E} \coloneqq \mathbb{E}'[y{\leftarrow}\mathbb{R}'']$ verifies the statement. If x does not occur in i_{s} then one uses Lemma H.22.1 to see i_{s} as a normal rigid multi context and reason as before. If x does not occur in u then one sees u as a normal and fine multi context with no holes, as all terms are external multi contexts. Note that x has to occur in u or i, because it occurs in t, and so the context $\mathbb{E}$ is always proper.

□

c) Stability of goodness by addition/removal of ES: The invariance of goodness for the $\triangleleft$-transitions requires (weak) goodness to be stable by addition appropriate ES next to the hole of K. Similarly, $\triangleright$-transitions require stability of (weak) goodness by removal of the innermost ES next to the hole in K. We first focus on adding ES, in the next two lemmas.

Lemma H.28 (Open goodness addition)**.** *Let e be open good and i be an inert term. Then:*

1) if $e(x)$ undefined or $e(x) = i$ then $[y{\leftarrow}x]e$ and $[y{\leftarrow}xz]e$ are open good.
2) $[y{\leftarrow}\lambda x.e']e$ is open good.

Proof.

1) We have to prove three facts:
   a) $\sigma_{[y\leftarrow x]e}$ *is a fireball substitution*. Three sub-cases:

- $y \in V_{\mathsf{cr}}$ and $e(x)$ undefined: then $\sigma_{[y\leftarrow x]e} =_{L.E.25.1} \{y\leftarrow x\} \cup \sigma_e$. By hypothesis, σ_e is a fireball substitution, and thus so is $\sigma_{[y\leftarrow x]e}$.
- $y \in V_{\mathsf{cr}}$ and $e(x) = i$: $\sigma_{[y\leftarrow x]e} =_{L.E.25.1} \{y\leftarrow \sigma_e(x)\} \cup \sigma_e$. By hypothesis, σ_e is a fireball substitution, thus $\sigma_e(x)$ is a fireball, and we conclude.
- $y \in V_{\mathsf{calc}}$: $\sigma_{[y\leftarrow x]e} =_{L.E.25.1} \sigma_e$, which by hypothesis is a fireball substitution.

b) $L_{[y\leftarrow x]e}$ *is a inert context*. Three sub-cases:

- $y \in V_{\mathsf{cr}}$: $L_{[y\leftarrow x]e} =_{L.E.25.2} L_e$, which by hypothesis is an inert context.
- $y \in V_{\mathsf{calc}}$ and $e(x)$ undefined: $L_{[y\leftarrow x]e} =_{L.E.25.2} [y\leftarrow x]L_e$. By hypothesis, L_e is an inert context, and thus so is $L_{[y\leftarrow x]e}$.
- $y \in V_{\mathsf{calc}}$ and $e(x) = i$: $L_{[y\leftarrow x]e} =_{L.E.25.2} [y\leftarrow \sigma_e(x)]L_e$. By hypothesis, L_e is an inert context, and we need to prove that $\sigma_e(x)$ is an inert term. Since e is open good, $\sigma_e(x)$ is a fireball, and if it is a value then $e(x)$ is also a value. By the side condition of the rule, $e(x)$ is an inert, and thus $\sigma_e(x)$ is not a value, that is, it is a inert term.

c) $[y\leftarrow x]e$ *has immediate values*. Assume that $\sigma_{[y\leftarrow x]e}(z)$ is a value for $z \neq \star$, we have to prove that $([y\leftarrow x]e)(z)$ is a value. Three sub-cases:

- $y \in V_{\mathsf{cr}}$ and $e(x)$ undefined: $\sigma_{[y\leftarrow x]e}(z) =_{L.E.25.1} (\{y\leftarrow x\} \cup \sigma_e)(z)$. If $z \neq y$ then it follows by the fact that e has immediate values (by hypothesis). Otherwise, $(\{y\leftarrow x\} \cup \sigma_e)(y) = x$ which is not a value, and so the statement trivially holds.
- $y \in V_{\mathsf{cr}}$ and $e(x) = i$: $\sigma_{[y\leftarrow x]e}(z) =_{L.E.25.1} (\{y\leftarrow \sigma_e(x)\} \cup \sigma_e)(z)$. If $z \neq y$ then it follows by the fact that e has immediate values (by hypothesis). Otherwise, $(\{y\leftarrow \sigma_e(x)\} \cup \sigma_e)(y) = \sigma_e(x)$. Since e has immediate values and $x \neq \star$ (because it occurs in $[y\leftarrow x]$), if $\sigma_e(x)$ is a value then $e(x)$ is a value, against the hypothesis that it is an inert term—then this case is not possible.
- $y \in V_{\mathsf{calc}}$: $\sigma_{[y\leftarrow x]e}(z) = \sigma_e(z)$ and the property follows from the fact that e has immediate value (by hypothesis).

2) We have to prove three facts:

a) $\sigma_{[y\leftarrow \lambda x.e']e}$ *is a fireball substitution*. Then $\sigma_{[y\leftarrow \lambda x.e']e} =_{L.E.25.1} \{y\leftarrow \lambda x.e'\downarrow\sigma_e\} \cup \sigma_e$. By hypothesis, σ_e is a fireball substitution, thus so is $\sigma_{[y\leftarrow \lambda x.e']e}$.

b) $L_{[y\leftarrow x]e}$ *is a inert context*. Then $L_{[y\leftarrow \lambda x.e']e} =_{L.E.25.2} L_e$, which by hypothesis is an inert context.

c) $[y\leftarrow \lambda x.e']e$ *has immediate values*. Assume that $\sigma_{[y\leftarrow \lambda x.e']e}(z)$ is a value for $z \neq \star$, we have to prove that $([y\leftarrow \lambda x.e']e)(z)$ is a value. Note that $\sigma_{[y\leftarrow \lambda x.e']e}(z) =_{L.E.25.1} (\{y\leftarrow \lambda x.e'\downarrow\sigma_e\} \cup \sigma_e)(z)$. If $z \neq y$ then it follows by the fact that e has immediate values (by hypothesis). Otherwise, $(\{y\leftarrow \lambda x.e'\downarrow\sigma_e\} \cup \sigma_e)(y) = \lambda x.e'\downarrow\sigma_e$ and $([y\leftarrow \lambda x.e']e)(y) = \lambda x.e'$ which, as required, is a value.

□

Lemma H.29 (Goodness addition)**.** *Let K be good and i be an inert term.*

1) If $e_K(x)$ undefined or $e_K(x) = i$ then $K\langle\langle\cdot\rangle[y\leftarrow x]\rangle$ and $K\langle\langle\cdot\rangle[y\leftarrow xz]\rangle$ are good.
2) If $y \notin \mathsf{fv}(F_K)$ then $K\langle\langle\cdot\rangle[y\leftarrow \lambda x.e']\rangle$ is good.

Proof. Let K' be either $K\langle\langle\cdot\rangle[y\leftarrow x]\rangle$, $K\langle\langle\cdot\rangle[y\leftarrow xz]\rangle$ or $K\langle\langle\cdot\rangle[y\leftarrow \lambda x.e']\rangle$ depending on which case we are proving. For both points, by Lemma H.28 $e_{K'}$ is open good and F_K is well-framed, so that we only have to show that the unfolding of the frame $F_{K'}\downarrow$ of K' is a fine and normal multi context compatible with $e_{K'}$ and that the unfolding $K'\downarrow$ of K' is fine.

- $F_{K'}\downarrow$ is a fine and normal multi context compatible with $e_{K'}$:
  1) We treat the case of $K\langle\langle\cdot\rangle[y\leftarrow x]\rangle$, for $K\langle\langle\cdot\rangle[w\leftarrow xz]\rangle$ the reasoning is identical. Note that $F_K = F_{K\langle\langle\cdot\rangle[y\leftarrow x]\rangle}$. So by *i.h.* we know that $F_K\downarrow$ is a fine and normal multi context. We only need to show that it is compatible with $\sigma_{[y\leftarrow x]e_K}$, knowing what we refer to as the *compatibility hypothesis*, that is, that it is compatible with $\sigma_{[y\leftarrow x]e_K}$. Three sub-cases:
     a) $y \in V_{\mathsf{cr}}$ and $e_K(x)$ undefined: then by Lemma E.25.1 we have $\sigma_{[y\leftarrow x]e_K} = \{y\leftarrow x\} \cup \sigma_{e_K}$. Since x is inert, compatibility then follows from the compatibility hypothesis.
     b) $y \in V_{\mathsf{cr}}$ and $e_K(x) = i$: then by Lemma E.25.1 we have $\sigma_{[y\leftarrow x]e_K} = \{y\leftarrow \sigma_{e_K}(x)\} \cup \sigma_{e_K}$. By hypothesis, $e_K(x)$ is an inert term. Since e_K is open good, it has immediate values, and so $\sigma_{e_K}(x)$ is an inert term as well because $x \neq \star$ because x occurs in $[y\leftarrow x]$. Compatibility then follows from the compatibility hypothesis.
     c) $y \in V_{\mathsf{calc}}$: then by Lemma E.25.2 we have $\sigma_{[y\leftarrow x]e_K} = \sigma_{e_K}$ and the property follows by compatibility hypothesis.
  2) Note that also in this case we have $F_K =_{L.H.5.2} F_{K\langle\langle\cdot\rangle[y\leftarrow \lambda x.e']\rangle}$, and the *i.h.* gives that it is a fine and normal multi context follows. We have to show it compatible with $\sigma_{[y\leftarrow x]e_K}$, knowing that it is compatible with σ_{e_K}. This is immediate, because by hypothesis $y \notin \mathsf{fv}(F_K)$ and thus $y \notin \mathsf{fv}(F_K\downarrow)$.
- $K'\downarrow$ is fine:

1) We treat the case of $K\langle\langle\cdot\rangle[y{\leftarrow}x]\rangle$, for $K\langle\langle\cdot\rangle[w{\leftarrow}xz]\rangle$ the reasoning is identical. We have $K\langle\langle\cdot\rangle[y{\leftarrow}x]\rangle{\downarrow} = K{\downarrow}\langle\langle\cdot\rangle[y{\leftarrow}x]{\downarrow}\sigma_{e_K}\rangle$ by Lemma X.2.3. Note that $\langle\cdot\rangle[y{\leftarrow}x]{\downarrow}$ is either the inert context $\langle\cdot\rangle$ or the inert context $\langle\cdot\rangle[y{\leftarrow}x]$. Now we apply various lemmas about multi contexts:
   - $\langle\cdot\rangle[y{\leftarrow}x]{\downarrow}$ is a fine multi context by Lemma H.22.2,
   - it is also is compatible with the fireball substitution σ_{e_K} because by hypothesis x is not bound to an abstraction in e_K, then
   - $\langle\cdot\rangle[y{\leftarrow}x]{\downarrow}\sigma_{e_K}$ is a fine context by Lemma H.24, and finally
   - $K{\downarrow}\langle\langle\cdot\rangle[y{\leftarrow}x]{\downarrow}\sigma_{e_K}\rangle$ is a fine context by Lemma H.26.1.
2) Trivial because by Lemma X.2.3 $K\langle\langle\cdot\rangle[y{\leftarrow}\lambda x.e']\rangle{\downarrow} = K{\downarrow}\langle\langle\cdot\rangle[y{\leftarrow}\lambda x.e']{\downarrow}\sigma_{e_K}\rangle = K{\downarrow}\langle\langle\cdot\rangle\sigma_{e_K}\rangle = K{\downarrow}$ that is fine by hypothesis because K is good.

□

Goodness is also stable under removal of ES. In order to show that, we first prove in the following lemma a corresponding property for multi contexts: rigidity, strength, and properness are stable under removal.

Lemma H.30. *Let $\mathbb{C}$ be a multi context.*

1) If $\mathbb{C}\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ is a rigid multi context then so is $\mathbb{C}$.
2) If $\mathbb{C}\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ is an external multi context then so is $\mathbb{C}$.

Moreover, if $\mathbb{C}\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ is proper then $\mathbb{C}$ is proper.

Proof. By induction on $\mathbb{C}$.

- *Empty*, *i.e.* $\mathbb{C} = \langle\cdot\rangle$.
  1) Then $\mathbb{C}\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle = \langle\cdot\rangle[x{\leftarrow}t]$ but no rigid multi context can have this shape, so this case is impossible.
  2) Then $\langle\cdot\rangle$ is an external multi context.
- *Variable*, *i.e.* $\mathbb{R} = x$.
  1) x is a rigid multi context.
  2) x is an external multi context.
- *Abstraction*, *i.e.* $\mathbb{C} = \lambda x.\mathbb{C}'$.
  1) Then $\mathbb{C}\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle = \lambda x.\mathbb{C}'\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ but no rigid multi context can have this shape, so this case is impossible.
  2) By *i.h.* $\mathbb{C}'$ is an external multi context, and then so is $\mathbb{C}$.
- *Application*, *i.e.* $\mathbb{C} = \mathbb{C}'\mathbb{C}''$.
  1) If $\mathbb{C}\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle = \mathbb{C}'\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle\mathbb{C}''\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ is a rigid multi context then $\mathbb{C}'\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ is a rigid multi context and $\mathbb{C}''\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ is an external multi context. By *i.h.*, $\mathbb{C}'$ is rigid and $\mathbb{C}''$ is strong. Then $\mathbb{C}$ is rigid.
  2) $\mathbb{C}\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ is an external multi context only if it is rigid. By Point 1, $\mathbb{C}$ is rigid, and thus strong.
- *Explicit substitution*, *i.e.* $\mathbb{C} = \mathbb{C}'[y{\leftarrow}\mathbb{C}'']$.
  1) If $\mathbb{C}\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle = \mathbb{C}'\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle[y{\leftarrow}\mathbb{C}''\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle]$ is a rigid multi context then $\mathbb{C}'\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ is a rigid multi context and $\mathbb{C}''\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ is a rigid multi context. By *i.h.*, both $\mathbb{C}'$ and $\mathbb{C}''$ are rigid, and then so is $\mathbb{C}$.
  2) If $\mathbb{C}\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle = \mathbb{C}'\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle[y{\leftarrow}\mathbb{C}''\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle]$ is an external multi context then $\mathbb{C}'\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ is an external multi context and $\mathbb{C}''\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle$ is a rigid multi context. By *i.h.*, $\mathbb{C}'$ is an external multi context and $\mathbb{C}''$ is a rigid multi context. Then $\mathbb{C}$ is an external multi context.

The moreover part follows evidently holds in the base cases, and it follows immediately from the *i.h.* in the inductive cases. □

The invariance of goodness by removals of the innermost ES next to the hole in K is simpler than stability for addition, and it is given by the next lemma.

Lemma H.31 (Goodness removal)**.**

1) If $[x{\leftarrow}b]e$ is open good then e is open good.
2) If $K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle$ is good, then K is good.

Proof.

1) We have to prove three facts:
   a) *σ_e is a fireball substitution*. By Lemma E.25.1, $\sigma_{[x\leftarrow b]e}$ is σ_e plus possibly a substitution on x. Therefore, if $\sigma_{[x\leftarrow b]e}$ is a fireball substitution then so is σ_e.
   b) *L_e is a inert context*. By Lemma E.25.2, $L_{[x\leftarrow b]e}$ is L_e plus possibly an ES on x. Therefore, if $L_{[x\leftarrow b]e}$ is a inert substitution context then so is L_e.

c) e *has immediate values*. By Lemma E.25.1, $\sigma_{[x\leftarrow b]e}$ is σ_e plus possibly a substitution on x. Therefore, if $\sigma_{[x\leftarrow b]e}$ has immediate values then so does σ_e.

2) By hypothesis $e_{K\langle\langle\cdot\rangle[x\leftarrow b]\rangle} =_{L.G.2.1} [x\leftarrow b]e_K$ is open good, and so e_K is open good by the previous point.
   a) We prove that F_K is well-framed w.r.t. σ_{e_K}. By hypothesis, $F_{K\langle\langle\cdot\rangle[x\leftarrow b]\rangle} =_{L.H.5.2} F_K$ is well-framed w.r.t. $\sigma_{e_{K\langle\langle\cdot\rangle[x\leftarrow b]\rangle}} = \sigma_{[x\leftarrow b]e_K}$. Now, by Lemma E.25.1, we have that $\sigma_{[x\leftarrow b]e_K}$ is σ_{e_K} plus possibly a substitution on x. Therefore, compatibility with respect to $\sigma_{[x\leftarrow b]e_K}$ implies compatibility with respect to σ_{e_K} and thus F_K is also well-framed w.r.t. σ_{e_K}.
   b) We prove $K\downarrow$ is a fine multi context. By hypothesis we know that $K\langle\langle\cdot\rangle[x\leftarrow b]\rangle$ is good and therefore that $K\langle\langle\cdot\rangle[x\leftarrow b]\rangle\downarrow =_{L.X.2.3} K\downarrow\langle\langle\cdot\rangle[x\leftarrow b]\downarrow\sigma_{e_K}\rangle$ is a fine multi context. Two cases:
      - If $b = v$ or $x \in V_{\mathsf{cr}}$ then $K\downarrow\langle\langle\cdot\rangle[x\leftarrow b]\downarrow\sigma_{e_K}\rangle = K\downarrow\langle\langle\cdot\rangle\{x\leftarrow b\downarrow\}\sigma_{e_K}\rangle = K\downarrow$, which is then a fine multi context.
      - Otherwise, $K\downarrow\langle\langle\cdot\rangle[x\leftarrow b]\downarrow\sigma_{e_K}\rangle = K\downarrow\langle\langle\cdot\rangle[x\leftarrow b\sigma_{e_K}]\rangle$. By Lemma H.30.2, $K\downarrow$ is a fine multi context. □

We now have all the ingredients to conclude that also Goodness is propagated.

Theorem H.32 (Goodness invariant)**.** *Let $s = e \bowtie K$ be a state reachable from an initial state s_0. Then s is good.*

Proof. By induction on the execution $\rho : s_0 \to^*_{\mathsf{M}} s$. If ρ is empty then $s = s_0$ and, by definition of initial state, $s_0 = e_0\triangleleft\langle\cdot\rangle$ for some well-named and pristine environment e_0: $e_0\triangleleft\langle\cdot\rangle$ is good by the definition of good, since e_0 is pristine.

If ρ is non-empty we look at the last transition $s' \to_{\mathsf{M}} s$, knowing by *i.h.* that the goodness invariant holds for s':

- $e[x\leftarrow y\,z]\triangleleft K \rightsquigarrow_{\beta_v} e([x\leftarrow b]e'\{w\leftarrow z\})\triangleleft K$ with $(e_K(y))^\alpha = \lambda w.([\star\leftarrow b]e')$ and $e_K(z) = v$ for some v.
  Goodness follows from the *i.h.*

- $e[x\leftarrow y\,z]\triangleleft K \rightsquigarrow_{\beta_i} e[x\leftarrow b]e'\triangleleft K\langle\langle\cdot\rangle[w\leftarrow z]\rangle$ with $(e_K(y))^\alpha = \lambda w.([\star\leftarrow b]e')$ and $e_K(z) = i$ for some inert term i.
  By *i.h.*, K is good; then $K\langle\langle\cdot\rangle[w\leftarrow z]\rangle$ is good by Lemma H.29.

- $e[x\leftarrow y]\triangleleft K \rightsquigarrow_{\mathsf{ren}} e\{x\leftarrow y\}\triangleleft K$ with $x \neq \star$.
  Goodness follows from the *i.h.*

- $e[x\leftarrow b]\triangleleft K \rightsquigarrow_{\mathsf{sea}_1} e\triangleleft K\langle\langle\cdot\rangle[x\leftarrow b]\rangle$ when b is an abstraction or when b is y or yz but y is not defined in e_K or $e_K(y)$ is not a value.
  To show that $K\langle\langle\cdot\rangle[x\leftarrow b]\rangle$ is good, note that by *i.h.*, K is good. For all cases but when b is an abstraction we can immediately apply Lemma H.29 and obtain that $K\langle\langle\cdot\rangle[x\leftarrow b]\rangle$ is good. When b is an abstraction, by *i.h.* we obtain that s' is well-named, and so $x \notin \mathsf{fv}(F_K)$. Then we can apply Lemma H.29 and obtain that $K\langle\langle\cdot\rangle[x\leftarrow b]\rangle$ is good.

- $\epsilon\triangleleft K \rightsquigarrow_{\mathsf{sea}_2} \epsilon\triangleright K$
  $\epsilon\triangleright K$ is obviously good because the property holds by *i.h.*

- $e\triangleright K\langle\langle\cdot\rangle[x\leftarrow b]\rangle \rightsquigarrow_{\mathsf{sea}_3} e[x\leftarrow b]\triangleright K$ where b is a variable or an application.
  To show that $e[x\leftarrow b]\triangleright K$ is good:
  – K *is good*: by *i.h.*, $K\langle\langle\cdot\rangle[x\leftarrow b]\rangle$ is good. By Lemma H.31, K is good.
  – $e[x\leftarrow b]\downarrow$ *is a strong fireball compatible with* σ_{e_K}: by *i.h.*, $e\downarrow$ is a strong fireball compatible with $\sigma_{e_{K\langle\langle\cdot\rangle[x\leftarrow b]\rangle}} =_{L.G.2.1} \sigma_{[x\leftarrow b]e_K}$. Two cases:
    ∗ $x \in V_{\mathsf{cr}}$: $e[x\leftarrow b]\downarrow = e\downarrow\{x\leftarrow b\downarrow\}$ and $\sigma_{[x\leftarrow b]e_K} =_{L.E.25.1} \{x\leftarrow\sigma_{e_K}(b)\}\cup\sigma_{e_K}$. Because $e\downarrow$ is a strong fireball (by *i.h.*) and $b\downarrow = b$ is a strong inert (by hypothesis of the transition), then $e\downarrow\{x\leftarrow b\downarrow\}$ is a strong fireball because strong fireballs are stable under the substitution of strong inerts. Let y be such that σ_{e_K} is a value. By compatibility of $e\downarrow$ with $\{x\leftarrow\sigma_{e_K}(b)\}\cup\sigma_{e_K}$, y occurs in $e\downarrow$ only as an argument, if ever. Two cases:
      · b is a variable z and $z \neq \star$ since it occurs in $[x\leftarrow b]$. Note that $z \neq y$, otherwise e_K would not have immediate values, against goodness of K, and that for the same reason one also has $x \neq y$. Then y does not occur as an argument in $e[x\leftarrow b]\downarrow = e\downarrow\{x\leftarrow z\}$.
      · b is an application zw. Note that if $z = y$ then $\sigma_{e_K}(b)$ is not a fireball, against the hypothesis that $\{x\leftarrow\sigma_{e_K}(b)\}\cup\sigma_{e_K}$ is a fireball substitution. Then y occurs only as an argument in $e[x\leftarrow b]\downarrow$, and compatibility holds.
    ∗ $x \in V_{\mathsf{calc}}$: then $e[x\leftarrow b]\downarrow = e\downarrow[x\leftarrow b]$ and $\sigma_{[x\leftarrow b]e_K} =_{L.E.25.1} \sigma_{e_K}$. Since $e\downarrow$ is a strong fireball (by *i.h.*) and b is a strong inert term (by hypothesis of the transition), then $e[x\leftarrow b]\downarrow$ is a strong fireball. Compatibility for $e\downarrow$ follows from the *i.h.*, we only have to analyze $[x\leftarrow b]$. Let y be such that σ_{e_K} is a value. Two cases:
      · b is a variable z and $z \neq \star$ since it occurs in $[x\leftarrow b]$. Note that $z \neq y$, otherwise e_K would not have immediate values, against goodness of K, and that for the same reason one also has $x \neq y$. Then compatibility holds.
      · b is an application zw. Note that if $z = y$ then $\sigma_{e_K}(b)$ is not a fireball, against the hypothesis that $\{x\leftarrow\sigma_{e_K}(b)\}\cup\sigma_{e_K}$ is a fireball substitution. Then $z \neq y$ and compatibility holds.

- $e \triangleright K\langle\langle\cdot\rangle[x \leftarrow v]\rangle \rightsquigarrow_{\mathsf{gc}} e \triangleright K$ with $x \notin \mathsf{fv}(e)$.
  By *i.h.*, $K\langle\langle\cdot\rangle[x \leftarrow v]\rangle$ is good. By Lemma H.31, K is good.

- $e \triangleright K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle \rightsquigarrow_{\mathsf{sea}_4} e'[x \leftarrow \lambda y.e] \triangleright K$.
  To show that $e'[x \leftarrow \lambda y.e] \triangleright K$ is good:
  - K *is good*: by *i.h.* $K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle$ is good. We have to prove that:
    * e_K *is open good*: note that $e_K =_{L.G.2.2} e_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}$ and $e_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}$ is open good because $K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle$ is good.
    * F_K *is well-framed w.r.t.* σ_{e_K}. By *i.h.* $F_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle} =_{L.H.5.1} F_K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle$ is well-framed w.r.t. $\sigma_{e_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}} =_{L.G.2.2} \sigma_{e_K}$. Therefore also F_K is well-framed w.r.t. σ_{e_K} by Lemma H.20.
    * $K\!\downarrow$ *is fine*: by hypothesis we know that $K\langle\langle\cdot\rangle[x \leftarrow \lambda y.e]\rangle$ is good and therefore that $K\langle\langle\cdot\rangle[x \leftarrow \lambda y.e]\rangle\!\downarrow =_{L.X.2.3} K\!\downarrow\langle\langle\cdot\rangle[x \leftarrow \lambda y.e]\!\downarrow\sigma_{e_K}\rangle = K\!\downarrow\langle\langle\cdot\rangle\{x \leftarrow \lambda y.e\!\downarrow\}\sigma_{e_K}\rangle = K\!\downarrow$ is a fine multi context.
  - $e'[x \leftarrow \lambda y.e]\!\downarrow$ *is a strong fireball compatible with* σ_{e_K}: since $e \triangleright K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle$ is good, we have that
    * $e\!\downarrow$ is a strong fireball compatible with $\sigma_{e_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}} =_{L.G.2.2} \sigma_{e_K}$.
    * $F_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}\!\downarrow =_{L.H.5.1} F_K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle\!\downarrow$ is a fine context compatible with $\sigma_{e_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}} =_{L.G.2.2} \sigma_{e_K}$.

    Then $e'[x \leftarrow \lambda y.e]\!\downarrow = F_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}\langle e\rangle\!\downarrow =_{L.H.6.1} F_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}\!\downarrow\langle e\!\downarrow\rangle$ is a strong fireball because $F_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}$ is normal by *i.h.*
    By Lemma H.25, compatibility of $F_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}\!\downarrow$ and $e\!\downarrow$ with σ_{e_K} implies compatibility of $F_{K\langle e'[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}\!\downarrow\langle e\!\downarrow\rangle$ with σ_{e_K}.

- $e \triangleright K\langle\langle\cdot\rangle[x \leftarrow \lambda y.e']\rangle \rightsquigarrow_{\mathsf{sea}_5} e' \triangleleft K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle$ with $x \in \mathsf{fv}(e)$.
  We need to prove that $e' \triangleleft K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle$ is good. By *i.h.*, $e \triangleright K\langle\langle\cdot\rangle[x \leftarrow \lambda y.e']\rangle$ is good, which means that:
  - $e_{K\langle\langle\cdot\rangle[x \leftarrow \lambda y.e']\rangle} =_{L.G.2.1} [x \leftarrow \lambda y.e']e_K$ is open good,
  - $F_{K\langle\langle\cdot\rangle[x \leftarrow \lambda y.e']\rangle}$ is well-framed w.r.t. $\sigma_{e_{K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}} =_{L.G.2.2} \sigma_{e_K}$. Therefore, by Lemma H.20, F_K is also well-framed w.r.t. σ_{e_K}.
  - $K\langle\langle\cdot\rangle[x \leftarrow \lambda y.e']\rangle\!\downarrow =_{L.X.2.3} K\!\downarrow\langle\langle\cdot\rangle[x \leftarrow \lambda y.e']\!\downarrow\sigma_{e_K}\rangle = K\!\downarrow\langle\langle\cdot\rangle\{x \leftarrow \lambda y.e'\!\downarrow\}\!\downarrow\sigma_{e_K}\rangle = K\!\downarrow$ is a fine multi context.
  - $e\!\downarrow$ is a strong fireball compatible with $\sigma_{[x \leftarrow \lambda y.e']e_K} =_{L.E.25.1} \{x \leftarrow \lambda y.e'\!\downarrow\sigma_{e_K}\}\sigma_{e_K}$,

  We have to prove that:
  - $K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle$ *is good*: that is, we have to prove that
    * $e_{K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}$ *is open good*: note that $e_{K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle} =_{L.G.2.2} e_K$. By *i.h.* (first item above), we know that $[x \leftarrow \lambda y.e']e_K$ is open good. Then, e_K is open good by the open goodness removal lemma (Lemma H.31).
    * $F_{K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}$ *is well-framed w.r.t.* $\sigma_{e_{K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}} =_{L.G.2.2} \sigma_{e_K}$: note that $F_{K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle} =_{L.H.5.1} F_K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle$ and that we already proved that F_K is well-framed w.r.t. σ_{e_K}. Therefore we just need to show $F_K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle\!\downarrow =_{L.H.6.2} F_K\!\downarrow\langle e\!\downarrow\{x \leftarrow \lambda y.\langle\cdot\rangle\}\rangle$ to be a normal fine multi-context compatible with σ_{e_K}.
      Since F_K is well-framed w.r.t. σ_{e_K}, $F_K\!\downarrow$ is a fine and normal multi context compatible with σ_{e_K}. By the fourth item of the *i.h.* above, $e\!\downarrow$ is a strong fireball compatible with $\{x \leftarrow \lambda y.e'\!\downarrow\sigma_{e_K}\}\sigma_{e_K}$. By the hypothesis on the transition, $x \in \mathsf{fv}(e)$, and by the garbage invariant $x \in \mathsf{fv}(e\!\downarrow)$. Note that the compatibility hypothesis on $e\!\downarrow$ implies in particular that $e\!\downarrow$ is compatible with $\{x \leftarrow v\}$.
      Then by Lemma H.27 there exists a fine and normal context $\mathbb{E}$ such that $x \notin \mathsf{fv}(\mathbb{E})$ and $e\!\downarrow = \mathbb{E}\langle x\rangle$, and (by the hypothesis on $e\!\downarrow$) $\mathbb{E}$ is compatible with σ_{e_K}. By Lemma H.26, $e\{x \leftarrow \lambda y.\langle\cdot\rangle\}\!\downarrow = \mathbb{E}\langle x\rangle\{x \leftarrow \lambda y.\langle\cdot\rangle\} =_{L.H.23} \mathbb{E}\langle\lambda y.\langle\cdot\rangle\rangle$ is a fine and normal context, compatible with σ_{e_K} because $\mathbb{E}$ is. Note that $F_{K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle}\!\downarrow = F_K\!\downarrow\langle e\!\downarrow\{x \leftarrow \lambda y.\langle\cdot\rangle\}\rangle = F_K\!\downarrow\langle\mathbb{E}\langle\lambda y.\langle\cdot\rangle\rangle\rangle$, which is finally proved to be a fine and normal context compatible with σ_{e_K} by applying Lemma H.26 once more.
    * $K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle\!\downarrow$ *is fine:* $K\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\rangle\!\downarrow =_{L.X.2.3} K\!\downarrow\langle e[x \leftarrow \lambda y.\langle\cdot\rangle]\!\downarrow\sigma_{e_K}\rangle = K\!\downarrow\langle\mathbb{E}\langle\lambda y.\langle\cdot\rangle\rangle\sigma_{e_K}\rangle$
      where $\mathbb{E}\langle\lambda y.\langle\cdot\rangle\rangle$ is the fine (and normal) context compatible with σ_{e_K} built in the previous proof item. Thus, by Lemma H.24, $\mathbb{E}\langle\lambda y.\langle\cdot\rangle\rangle\sigma_{e_K}$ is a fine multi context and by Lemma H.26.1 so is $K\!\downarrow\langle\mathbb{E}\langle\lambda y.\langle\cdot\rangle\rangle\sigma_{e_K}\rangle$.

□

We conclude this section with the following theorem, proving the property of machine contexts that motivated the introduction of the Good invariant in the first place:

Theorem H.33 (Contextual read-back)**.** *Let $s = e \bowtie K$ be a reachable state. Then $K\!\downarrow$ is a proper external multi context.*

See p. 12 Thm. X.3

Proof. By Thm. H.32 s is good. Then $K\!\downarrow$ is a proper external multi context. □

F. Proof of the SCAM Implementation Theorem

Here we are supposed to prove the following theorem.

Theorem H.34 (SCAM implementation)**.**

1) Relaxed β-projection*: let s be a reachable state. If $s \rightsquigarrow_{\beta_v} s'$ then $s{\downarrow}(\rightarrow_{\mathsf{xm}}\rightarrow_{\mathsf{xe}})^{+}s'{\downarrow}$, and if $s \rightsquigarrow_{\beta_i} s'$ then $s{\downarrow} \rightarrow^{+}_{\mathsf{xm}}\equiv s'{\downarrow}$.*

2) Strong implementation*: the SCAM is a relaxed implementation of the strategy $(\rightarrow_{\mathsf{x}}, \equiv)$.*

See p. 12 Thm. X.4

We prove the two points separately.

G. Proof of Relaxed Projection

In this section we provide the proof of Theorem X.4 (proved in Thm. H.34): the read back projects β transitions to steps in the calculus. More precisely, each $\rightsquigarrow_{\beta_v}$ transition projects to one or more $\rightarrow_{\mathsf{xm}}$ steps followed by as many $\rightarrow_{\mathsf{xe}}$ steps — Theorem X.4.1 (proved in Thm. H.36) —, and each $\rightsquigarrow_{\beta_i}$ transition projects to one or more $\rightarrow_{\mathsf{xm}}$ steps *up to structural equivalence*, *i.e.* $\rightarrow^{+}_{\mathsf{xm}}\equiv$ (hence the term "relaxed") — Theorem X.4.2 (proved in Thm. H.38).

For this reason, we first need an auxiliary lemma that shows how substitutions can be permutated in terms. The following lemma proves (under suitable requirements on variables) that substitution contexts commute with open contexts, up to the structural equality of VSC.

Lemma H.35 (Open and substitution contexts commute up to $\equiv$)**.** *Let L be a substitution context and O be an open context such that:*

- $\mathsf{dom}(L) \perp \mathsf{fv}(O)$,
- $\mathsf{fv}(L) \perp \mathsf{dom}(O)$,
- $\mathsf{dom}(L) \perp \mathsf{dom}(O)$.

Then $L\langle O\langle t\rangle\rangle \equiv O\langle L\langle t\rangle\rangle$.

Proof. By induction on the structure of O:

- If $O = \langle\cdot\rangle$, the statement follows trivially.
- If $O = O'u$, then $L\langle O\langle t\rangle\rangle = L\langle O'\langle t\rangle u\rangle$. We prove that $L\langle O'\langle t\rangle u\rangle \equiv^{*}_{@\mathsf{l}} L\langle O'\langle t\rangle\rangle u$ by induction on the structure of L:
  - The case $L = \langle\cdot\rangle$ is trivial.
  - When $L = L'[y{\leftarrow}p]$, $L\langle O'\langle t\rangle u\rangle = L'\langle O'\langle t\rangle u\rangle[y{\leftarrow}p]$. By *i.h.* $L'\langle O'\langle t\rangle u\rangle[y{\leftarrow}p] \equiv^{*}_{@\mathsf{l}} L'\langle O'\langle t\rangle\rangle u[y{\leftarrow}p]$ (note that the additional requirement that $y \notin \mathsf{dom}(O')$ follows from the hypothesis that $\mathsf{dom}(L) \perp \mathsf{dom}(O)$). In order to apply one last time $\equiv_{@\mathsf{l}}$ and conclude, we need to show that $y \notin \mathsf{fv}(u)$, which follows from the hypothesis that $\mathsf{dom}(L) \perp \mathsf{fv}(O)$.

  Finally, we conclude by using the *i.h.*
- The case $O = uO'$ is similar to the case above.
- If $O = O'[x{\leftarrow}u]$ then $L\langle O\langle t\rangle\rangle = L\langle O'[x{\leftarrow}u]\rangle$. We prove that $L\langle O'[x{\leftarrow}u]\rangle \equiv^{*}_{\mathsf{com}} L\langle O'\rangle[x{\leftarrow}u]$ by induction on the structure of L:
  - The case $L = \langle\cdot\rangle$ is trivial.
  - When $L = L'[y{\leftarrow}p]$, $L\langle O'[x{\leftarrow}u]\rangle = L'\langle O'[x{\leftarrow}u]\rangle[y{\leftarrow}p]$. By *i.h.* $L'\langle O'[x{\leftarrow}u]\rangle[y{\leftarrow}p] \equiv^{*}_{\mathsf{com}} L'\langle O'\rangle[x{\leftarrow}u][y{\leftarrow}p]$ (again, the additional requirement that $y \notin \mathsf{dom}(O')$ follows from the hypothesis that $\mathsf{dom}(L) \perp \mathsf{dom}(O)$). In order to apply one last time $\equiv_{\mathsf{com}}$ and conclude, we need to show that $y \notin \mathsf{fv}(u)$ and $x \notin \mathsf{fv}(p)$: the first follows from the hypothesis that $\mathsf{dom}(L) \perp \mathsf{fv}(O)$, the second from $\mathsf{dom}(O) \perp \mathsf{fv}(L)$.

  Finally, we conclude by using the *i.h.*
- If $O = u[x{\leftarrow}O']$ then $L\langle O\langle t\rangle\rangle = L\langle u[x{\leftarrow}O'\langle t\rangle]\rangle$. We prove that $L\langle u[x{\leftarrow}O'\langle t\rangle]\rangle \equiv^{*}_{[\cdot]} u[x{\leftarrow}L\langle O'\langle t\rangle\rangle]$ by induction on the structure of L:
  - The case $L = \langle\cdot\rangle$ is trivial.
  - When $L = L'[y{\leftarrow}p]$, $L\langle u[x{\leftarrow}O'\langle t\rangle]\rangle = L'\langle u[x{\leftarrow}O'\langle t\rangle]\rangle[y{\leftarrow}p]$. By *i.h.* $L'\langle u[x{\leftarrow}O'\langle t\rangle]\rangle[y{\leftarrow}p] \equiv^{*}_{[\cdot]} u[x{\leftarrow}L'\langle O'\langle t\rangle\rangle][y{\leftarrow}p]$ (again, the additional requirement that $y \notin \mathsf{dom}(O')$ follows from the hypothesis that $\mathsf{dom}(L) \perp \mathsf{dom}(O)$). In order to apply one last time $\equiv_{[\cdot]}$ and conclude, we need to show that $y \notin \mathsf{fv}(u)$, which follows from the hypothesis that $\mathsf{dom}(L) \perp \mathsf{fv}(O)$ and $\mathsf{dom}(L) \perp \mathsf{dom}(O)$.

  Finally, we conclude by using the *i.h.* □

Theorem H.36 (Relaxed projection)**.** *Let s be a reachable state.*

1) If $s \rightsquigarrow_{\beta_v} s'$ then $s{\downarrow}\ (\rightarrow_{\mathsf{xm}}\rightarrow_{\mathsf{xe}})^{+}\ s'{\downarrow}$.

2) If $s \rightsquigarrow_{\beta_i} s'$ then $s{\downarrow} \rightarrow^{+}_{\mathsf{xm}}\equiv s'{\downarrow}$.

See p. 12 Theorem X.4.

Proof. The state s must be $e[x{\leftarrow}y\ z]\triangleleft K$ with $(e_K(y))^{\alpha} = \lambda w.([\star{\leftarrow}b]e'')$. There are two cases, depending on whether $e_K(z)$ is an abstraction or an inert.

- *Abstraction*, *i.e.* $e_K(z) = v$. Since s is reachable it is well-named by Thm. H.8 and thus by Lemma G.1.5 K is well-named and so by Lemma G.1.3 e_K is well named and thus by Lemma E.26 $z\sigma_{e_K} = \sigma_{e_K}(z)$ is a value. The machine transition is

$$s = e[x{\leftarrow}y\ z]\triangleleft K \rightsquigarrow_{\beta_v} e([x{\leftarrow}b]e''\{w{\leftarrow}z\})\triangleleft K = s'$$

and it projects as follows:

$$\begin{array}{lll}
s\downarrow & = & (e[x\leftarrow y\,z]\triangleleft K)\downarrow \\
& = & K\downarrow\langle(e[x\leftarrow y\,z])\downarrow\sigma_{e_K}\rangle \\
& =_{K\ good} & \mathbb{E}\langle(e[x\leftarrow y\,z])\downarrow\sigma_{e_K}\rangle \\
& =_{L.E.19} & \mathbb{E}\langle O\langle[\star\leftarrow yz]\downarrow\sigma_{e_K}\rangle\rangle \\
& = & \mathbb{E}\langle O\langle y\sigma_{e_K}\ z\sigma_{e_K}\rangle\rangle \\
& = & \mathbb{E}\langle O\langle(\lambda w.([\star\leftarrow b]e'')\downarrow\sigma_{e_K})\ (z\sigma_{e_K})\rangle\rangle \\
& (\rightarrow_{\mathsf{xm}}\rightarrow_{\mathsf{xe}})^+_{\substack{L.X.1,\\ z\sigma_{e_K}\ \textit{is a value}}} & \mathbb{E}\langle O\langle([\star\leftarrow b]e'')\downarrow\sigma_{e_K}\{w\leftarrow z\sigma_{e_K}\}\rangle\rangle \\
& = & \mathbb{E}\langle O\langle([\star\leftarrow b]e'')\downarrow\{w\leftarrow z\}\sigma_{e_K}\rangle\rangle \\
& =_{L.E.18.2} & \mathbb{E}\langle O\langle([\star\leftarrow b]e''\{w\leftarrow z\})\downarrow\sigma_{e_K}\rangle\rangle \\
& = & \mathbb{E}\langle O\langle([\star\leftarrow b\{w\leftarrow z\}](e''\{w\leftarrow z\}))\downarrow\sigma_{e_K}\rangle\rangle \\
& =_{L.E.19} & \mathbb{E}\langle(e[x\leftarrow b\{w\leftarrow z\}](e''\{w\leftarrow z\}))\downarrow\sigma_{e_K}\rangle \\
& = & \mathbb{E}\langle(e([x\leftarrow b]e''\{w\leftarrow z\}))\downarrow\sigma_{e_K}\rangle \\
& =_{K\ good} & K\downarrow\langle(e([x\leftarrow b]e''\{w\leftarrow z\}))\downarrow\sigma_{e_K}\rangle \\
& = & (e([x\leftarrow b]e''\{w\leftarrow z\})\triangleleft K)\downarrow \qquad = \quad s'\downarrow \\
& = & s'\downarrow
\end{array}$$

- *Inert*, *i.e.* $e_K(z) = i$. Since s is reachable it is good by Thm. H.32 and so σ_{e_K} has immediate values. Thus, since $z \neq \star$ because it occurs in $[x\leftarrow yz]$ and since $e_K(z)$ is an inert, $z\sigma_{e_K} = \sigma_{e_K}(z)$ must be an inert by definition of the immediate values property. The machine transition is

$$s = e[x\leftarrow y\,z]\triangleleft K \rightsquigarrow_{\beta_i} e[x\leftarrow b]e''\triangleleft K\langle\langle\cdot\rangle[w\leftarrow z]\rangle = s'$$

and it projects as follows:

$$\begin{array}{lll}
s\downarrow & = & (e[x\leftarrow y\,z]\triangleleft K)\downarrow \\
& = & K\downarrow\langle(e[x\leftarrow y\,z])\downarrow\sigma_{e_K}\rangle \\
& =_{K\ good} & \mathbb{E}\langle(e[x\leftarrow y\,z])\downarrow\sigma_{e_K}\rangle \\
& =_{L.E.19} & \mathbb{E}\langle O\langle[\star\leftarrow yz]\downarrow\sigma_{e_K}\rangle\rangle \\
& = & \mathbb{E}\langle O\langle y\sigma_{e_K}\ z\sigma_{e_K}\rangle\rangle \\
& = & \mathbb{E}\langle O\langle(\lambda w.([\star\leftarrow b]e'')\downarrow\sigma_{e_K})\ (z\sigma_{e_K})\rangle\rangle \\
& \rightarrow^+_{\mathsf{xm}\ \substack{L.X.1,\\ z\sigma_{e_K}\ \textit{is an inert}}} & \mathbb{E}\langle O\langle([\star\leftarrow b]e'')\downarrow\sigma_{e_K}[w\leftarrow z\sigma_{e_K}]\rangle\rangle \\
& \equiv_{L.H.35} & \mathbb{E}\langle O\langle([\star\leftarrow b]e'')\downarrow\sigma_{e_K}\rangle[w\leftarrow z\sigma_{e_K}]\rangle \\
& =_{L.E.19} & \mathbb{E}\langle e[x\leftarrow b]e''\downarrow\sigma_{e_K}[w\leftarrow z\sigma_{e_K}]\rangle \\
& = & \mathbb{E}\langle e[x\leftarrow b]e''\downarrow[w\leftarrow z]\sigma_{e_K}\rangle \\
& = & \mathbb{E}\langle(e[x\leftarrow b]e''[w\leftarrow z])\downarrow\sigma_{e_K}\rangle \\
& =_{K\ good} & K\downarrow\langle(e[x\leftarrow b]e''[w\leftarrow z])\downarrow\sigma_{e_K}\rangle \\
& = & (e[x\leftarrow b]e''\triangleleft K\langle\langle\cdot\rangle[w\leftarrow z]\rangle)\downarrow \\
& = & s'\downarrow
\end{array}$$

□

H. Proof of overhead transparency

In this section we provide the proof of Lemma H.37, *i.e.* that the read back projects overhead transitions to equality in the VSC calculus.

Lemma H.37 (Overhead transparency)**.** *Let s be reachable. If $s \rightsquigarrow s'$ with a non-β transition, then $s\downarrow \equiv s'\downarrow$.*

Proof. By inspection of the non-multiplicative machine transitions. The proof for all the search steps is trivial since a search step has the form $e_1 \bowtie K_1 \rightarrow e_2 \bowtie K_2$ with $K_1\langle e_1\rangle = K_2\langle e_2\rangle$ and thus $s\downarrow = K_1\langle e_1\rangle\downarrow = K_2\langle e_2\rangle\downarrow = s'\downarrow$. We analyze the remaining steps.

- Case $e[x\leftarrow y]\triangleleft K \rightsquigarrow_{\mathsf{ren}} e\{x\leftarrow y\}\triangleleft K$ with $x \neq \star$. Since $e[x\leftarrow y]\triangleleft K$ is reachable, it is well-named and therefore, by Lemma G.1.4, $e[x\leftarrow y]$ is well-named and thus, by Lemma E.14, e is well-named and $y \notin \mathsf{bv}(e)$.

$$\begin{array}{ll} & e[x\leftarrow y]\triangleleft K\downarrow \\ = & K\langle e[x\leftarrow y]\rangle\downarrow \\ =_{L.X.2.3} & K\downarrow\langle e[x\leftarrow y]\downarrow\sigma_{e_K}\rangle \\ =_{x\in V_{cr}} & K\downarrow\langle e\downarrow\{x\leftarrow y\}\sigma_{e_K}\rangle \\ =_{L.E.18.2} & K\downarrow\langle e\{x\leftarrow y\}\downarrow\sigma_{e_K}\rangle \\ =_{L.X.2.3} & K\langle e\{x\leftarrow y\}\rangle\downarrow \\ = & e\{x\leftarrow y\}\triangleleft K\downarrow \end{array}$$

- Case $e\triangleright K\langle\langle\cdot\rangle[x\leftarrow v]\rangle \rightsquigarrow_{\mathsf{gc}} e\triangleright K$ with $x \notin \mathsf{fv}(e)$.

$$\begin{array}{ll} & e\triangleright K\langle\langle\cdot\rangle[x\leftarrow v]\rangle\downarrow \\ = & K\langle e[x\leftarrow v]\rangle\downarrow \\ =_{L.X.2.3} & K\downarrow\langle e[x\leftarrow v]\downarrow\sigma_{e_K}\rangle \\ = & K\downarrow\langle e\downarrow\{x\leftarrow v\downarrow\}\sigma_{e_K}\rangle \\ =_{L.E.18.3,\, x\notin\mathsf{fv}(e)} & K\downarrow\langle e\downarrow\sigma_{e_K}\rangle \\ =_{L.X.2.3} & K\langle e\rangle\downarrow \\ = & e\triangleright K\downarrow \end{array}$$

□

We now have all the ingredients to prove Theorem X.4.2 (proved in Thm. H.38), *i.e.* the implementation theorem for the SCAM machine:

Theorem H.38 (Strong CbV Implementation)**.** *The SCAM is a relaxed implementation of the strong fireball strategy* $(\to_{\mathsf{x}}, \equiv)$.

See p. 12 Theorem X.4.

Proof. The strong fireball strategy $(\to_{\mathsf{x}}, \equiv)$ is a structural strategy by Prop. C.4. We show that $(\rightsquigarrow_{SCAM}, \to_{\mathsf{x}}, \equiv, \cdot\downarrow)$ form a relaxed implementation system, and obtain the statement by Thm. V.3. First of all, the initialization constraint for the SCAM is given by Lemma VII.4. About the conditions for a relaxed implementation system:

1) *Relaxed β-projection:* by Theorem X.4.1 (proved in Thm. H.36).
2) *Overhead transparency:* by Lemma H.37.
3) *Overhead transitions terminates:* it follows by Corollary XI.5.
4) *Halt:* by inspection of the SCAM transitions, the only normal states of the machine have the form $e\triangleright$. Since this state is good, $e\downarrow$ is a strong fireball. By Lemma II.5, $e\downarrow$ is a $\to_{\mathsf{x}}$-normal form.
5) *Relaxed determinism*:
   - $\to_{\mathsf{x}}$ *is diamond*: by Prop. III.1.1.
   - $\rightsquigarrow_{SCAM}$ *is deterministic*: by a simple inspection of the transitions and the well-naming property. Well-naming grants uniqueness of lookup in the environment during transitions $\rightsquigarrow_{\beta_v}$ and $\rightsquigarrow_{\beta_i}$.

□

Appendix I
Proofs of Section XI (Complexity)

In this section we prove that the SCAM can be implemented within a bilinear time overhead. The fundamental invariant is the *size* invariant, proved in Thm. I.3: it basically shows that the size of the abstractions present in the unevaluated parts of a reachable state is bound by the size of the initial state.

First of all, we show that the measure of the initial state is linearly related to the size of the initial λ-term:

Lemma I.1 (Linear compilation)**.** *Let t be a λ-term. Then $|\underline{t}| \le 2|t|$.*

See p. 12 Lemma XI.1

Proof. We prove this statement mutually with the corresponding statement for the auxiliary translation, *i.e.* that $|e| \le 2|t|$ when $(\_, e) := \overline{t}$. We proceed by induction on the structure of t:

- If $t = x$, then $\underline{t} = [\star\leftarrow x]$, and $|\underline{t}| = 2 = 2|t|$.
- If $t = \lambda x.u$, then $\underline{t} = [\star\leftarrow\lambda x.\underline{u}]$, and $|\underline{t}| = 2 + |\underline{u}| \le_{i.h.} 2 + 2|u| = 2(|u| + 1) = 2|t|$.
- If $t = up$, then $\underline{t} = [\star\leftarrow xy]ee'$ where $(x, e) := \underline{u}$ and $(y, e') := \underline{p}$. By *i.h.* $|e| \le 2|u|$ and $|e'| \le 2|p|$. Hence $|\underline{t}| = |[\star\leftarrow xy]ee'| = 2 + |e| + |e'| \le_{i.h.} 2 + 2|u| + 2|p| = 2(|u| + |p| + 1) = 2|t|$.

Concerning the auxiliary translation:

- If $t = x$, then $\overline{t} = (x, \epsilon)$, and $|\epsilon| = 0 < 2 = 2|t|$.
- If $t = \lambda x.u$, then $\overline{t} = (z, [z\leftarrow\lambda x.\underline{u}])$, and $|e| = 2 + |\underline{u}| \le_{i.h.} 2 + 2|u| = 2|t|$.

- If $t = up$, then $\bar{t} = (z, [z{\leftarrow}xy]ee')$ where $(x, e) := \underline{u}$ and $(y, e') := \underline{p}$. By *i.h.* $|e| \leq 2|u|$ and $|e'| \leq 2|p|$. Hence $|[z{\leftarrow}xy]ee'| = 2 + |e| + |e'| \leq_{i.h.} 2 + 2|u| + 2|p| = 2|t|$. □

In the proof of the size invariant we shall use repeatedly the following trivial properties of size:

Lemma I.2 (Properties of $|\cdot|$)**.** *For all environments e, e':*

1) $|ee'| = |e| + |e'|$
2) $|e\{x{\leftarrow}y\}| = |e|$
3) *if* $e =_\alpha e'$*, then* $|e| = |e'|$
4) $|[x{\leftarrow}b]e'| = |[\star{\leftarrow}b]e'|$

Proof.

1) By induction on the structure of e':
   - Case ϵ: $|e\epsilon| = |e| = |e| + 0 = |e| + |\epsilon|$.
   - Case $e''[x{\leftarrow}b]$: $|ee'| = |ee''[x{\leftarrow}b]| = 1 + |ee''| + |b| =_{i.h.} 1 + |e| + |e''| + |b| = |e| + |e''[x{\leftarrow}b]| = |e| + |e'|$.
2) Obvious because the definition of $|\cdot|$ does not care about names.
3) Obvious because the definition of $|\cdot|$ does not care about names.
4) Obvious because the definition of $|\cdot|$ does not care about names. □

We turn to the proof of the size invariant:

Theorem I.3 (Size invariant)**.** *Let $s = e \bowtie K$ be a state reachable starting from an initial state $s_0 = e_0 \triangleleft \langle\cdot\rangle$. Then $|v| \leq |e_0|$ holds for every abstraction v either in e (when the state is $e \triangleleft K$) or in e_K.*

Proof. By induction on the execution $\rho : s_0 \to_{\mathsf{M}}^* s$.

If ρ is empty then $s = s_0$; in this case, the size invariant is trivial by the definition of size.

If ρ is non-empty we look at the last transition $s' \to_{\mathsf{M}} s$, knowing by *i.h.* that the size invariant holds for s':

- $e[x{\leftarrow}y\,z]\triangleleft K \rightsquigarrow_{\beta_v} e([x{\leftarrow}b]e'\{w{\leftarrow}z\})\triangleleft K$ with $(e_K(y))^\alpha = \lambda w.([\star{\leftarrow}b]e')$ and $e_K(z) = v$ for some v. Every value v in e_K is such that $|v| \leq |e_0|$ because the property holds by *i.h.*. Moreover, values in $e([x{\leftarrow}b]e'\{w{\leftarrow}z\})$ are either renamings of α-renamings of values in e_K or values in e and thus in $e[x{\leftarrow}y\,z]$. Therefore the property holds by *i.h.*, Lemma I.2.2 and Lemma I.2.3.
- $e[x{\leftarrow}y\,z]\triangleleft K \rightsquigarrow_{\beta_i} e[x{\leftarrow}b]e'\triangleleft K\langle\langle\cdot\rangle[w{\leftarrow}z]\rangle$ with $(e_K(y))^\alpha = \lambda w.([\star{\leftarrow}b]e')$ and $e_K(z) = i$ for some inert term i. Every value v in $e_{K\langle\langle\cdot\rangle[w{\leftarrow}z]\rangle} =_{L.G.2.1} [w{\leftarrow}z]e_K$ is such that $|v| \leq |e_0|$ because the property holds by *i.h.* for e_K. Moreover, values in $e[x{\leftarrow}b]e'$ are either α-renamings of values in e_K or values in e and thus in $e[x{\leftarrow}y\,z]$. Therefore the property holds by *i.h.* and Lemma I.2.3.
- $e[x{\leftarrow}y]\triangleleft K \rightsquigarrow_{\mathsf{ren}} e\{x{\leftarrow}y\}\triangleleft K$ with $x \neq \star$. Every abstraction in e_K or in $e\{x{\leftarrow}y\}$ is an abstraction in e_K or a renaming of an abstraction in $e[x{\leftarrow}y]$. The property follows from *i.h.* by Lemma I.2.2.
- $e[x{\leftarrow}b]\triangleleft K \rightsquigarrow_{\mathsf{sea}_1} e\triangleleft K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle$ when b is an abstraction or when b is y or yz but y is not defined in e_K or $e_K(y)$ is not a value. Every abstraction in e or in $e_{K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle} =_{L.G.2.1} [x{\leftarrow}b]e_K$ is also in $e[x{\leftarrow}b]$ or in e_K. Therefore the property holds by *i.h.*.
- $\epsilon\triangleleft K \rightsquigarrow_{\mathsf{sea}_2} \epsilon\triangleright K$. Immediate because the context does not change and the environment is empty both before and after the transition.
- $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle \rightsquigarrow_{\mathsf{sea}_3} e[x{\leftarrow}b]\triangleright K$ where b is a variable or an application. Every abstraction in e_K is also in $e_{K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle} =_{L.G.2.1} [x{\leftarrow}b]e_K$. Therefore the property holds by *i.h.*.
- $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle \rightsquigarrow_{\mathsf{gc}} e\triangleright K$ with $x \notin \mathsf{fv}(e)$. Every abstraction in e_K is also in $e_{K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle} =_{L.G.2.1} [x{\leftarrow}v]e_K$. Therefore the property holds by *i.h.*.
- $e\triangleright K\langle e'[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle \rightsquigarrow_{\mathsf{sea}_4} e'[x{\leftarrow}\lambda y.e]\triangleright K$. Trivial because $e_{K\langle e'[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle} =_{L.G.2.2} e_K$.
- $e\triangleright K\langle\langle\cdot\rangle[x{\leftarrow}\lambda y.e']\rangle \rightsquigarrow_{\mathsf{sea}_5} e'\triangleleft K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle$ with $x \in \mathsf{fv}(e)$. Every abstraction in e' or in $e_{K\langle e[x{\leftarrow}\lambda y.\langle\cdot\rangle]\rangle} =_{L.G.2.2} e_K$ is also in $e_{K\langle\langle\cdot\rangle[x{\leftarrow}\lambda y.e']\rangle} =_{L.G.2.1} [x{\leftarrow}\lambda y.e']e_K$. Therefore the property holds by *i.h.*. □

A. Number of overhead transitions

The aim of this sub-section is to provide a bound on the number of machine steps as a function of the number of β-steps, *i.e.* Corollary XI.5 (proved in Corollary I.7). The result is obtained as a corollary of Lemma XI.3 (proved in Lemma I.4) and Lemma XI.4 (proved in Lemma I.6).

We estimate the number of overhead transitions in a modular way: first bounding those in all strong phases, then those in the open phases.

The number of transitions of strong phases is bound by the number of transitions of open phases: we provide a global analysis in the following lemma.

Lemma I.4 (Open phases bound strong phases)**.** *Let $\rho\colon s_0 \rightsquigarrow s$ an execution of the SCAM. Then:*

See p. 12 Lemma XI.3

$$|\rho|_{\mathsf{sea}_2} + |\rho|_{\mathsf{sea}_3} + |\rho|_{\mathsf{gc}} + |\rho|_{\mathsf{sea}_4} + |\rho|_{\mathsf{sea}_5} \leq |\rho|_{\beta_i} + 4|\rho|_{\mathsf{sea}_1} + 1.$$

Proof. The statement is a consequence of the following inequalities:

- $|\rho|_{\mathsf{sea}_2} \leq |\rho|_{\mathsf{sea}_5} + 1$ because only sea_5 switches the phase to $\triangleleft$ (plus 1 because the initial state is in the $\triangleleft$ phase).
- $|\rho|_{\mathsf{sea}_3} + |\rho|_{\mathsf{gc}} \leq |\rho|_{\beta_i} + |\rho|_{\mathsf{sea}_1}$ because sea_3 and gc pop entries of the form $[x{\leftarrow}b]$ from the machine environment, which are pushed only by β_i and sea_1.
- $|\rho|_{\mathsf{sea}_4} \leq |\rho|_{\mathsf{sea}_5}$ because sea_4 pops entries of the form $[x{\leftarrow}\lambda y.\langle\cdot\rangle]$ from the machine environment, which are pushed only by sea_5.
- $|\rho|_{\mathsf{sea}_5} \leq |\rho|_{\mathsf{sea}_1}$ because sea_5 pops entries of the form $[x{\leftarrow}\lambda y.e']$ from the machine environment, which are pushed only by sea_1.

□

To bound the number of overhead transitions of open phases, we introduce a new measure $\|\cdot\|$ over machine states:

$$\begin{array}{lcccc} \textsc{Environments} & & |\epsilon| \coloneqq 0 & & |e[x{\leftarrow}b]| \coloneqq 1 + |e| + |b| \\ \textsc{Contexts} & \|\langle\cdot\rangle\| \coloneqq 0 & \|e[x{\leftarrow}\lambda y.K]\| \coloneqq \|K\| & & \|K[x{\leftarrow}b]\| \coloneqq \|K\| + |b| \\ \textsc{States} & & \|e \triangleleft K\| \coloneqq |e| + \|K\| & & \|e \triangleright K\| \coloneqq \|K\| \end{array}$$

Note: the measure $\|\cdot\|$ completely ignores the environment e on the left of the cursor during the strong phase (case $\|e \triangleright K\|$). In fact, that environment has already been fully evaluated, and thus it should not contribute in any way. The environment e in $\|e \triangleleft K\|$, instead, contributes with its size: in this way, the measure strictly decreases after an open overhead transition, since $\rightsquigarrow_{\mathsf{ren}}$ and sea_1 pop an entry from the environment on the left of the cursor.

First, we prove a couple of trivial properties of the measure, that will be used repeatedly in the proof of Lemma I.6.

Lemma I.5 (Properties of $\|\cdot\|$)**.** *For all contexts K, K':*

1) $\|K\langle K'\rangle\| = \|K\| + \|K'\|$
2) $\|K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle\| = \|K\| + |b|$

Proof.

1) By induction on the structure of K:
   - Case $\langle\cdot\rangle$: trivial.
   - Case $K[x{\leftarrow}b]$: $\|K''[x{\leftarrow}b]\langle K'\rangle\| = \|K''\langle K'\rangle[x{\leftarrow}b]\| = \|K''\langle K'\rangle\| + |b| =_{\textit{i.h.}} \|K''\| + \|K'\| + |b| = \|K''[x{\leftarrow}b]\| + \|K'\|$.
   - Case $e[x{\leftarrow}\lambda y.K'']$: $\|e[x{\leftarrow}\lambda y.K'']\langle K'\rangle\| = \|e[x{\leftarrow}\lambda y.K''\langle K'\rangle]\| = \|K''\langle K'\rangle\| =_{\textit{i.h.}} \|K''\| + \|K'\| = \|e[x{\leftarrow}\lambda y.K'']\| + \|K'\|$.
2) $\|K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle\| =_{P.1} \|K\| + \|\langle\cdot\rangle[x{\leftarrow}b]\| = \|K\| + |b|$.

□

Lemma I.6 (Measure during execution)**.** *Let s be a state reachable from s_0, and $s \rightsquigarrow_a s'$.*

See p. 12 Lemma XI.4

- *Beta transitions increase the measure: if $a \in \{\beta_\lambda, \beta_i\}$ then $\|s'\| \leq \|s\| + \|s_0\|$.*
- *Open overhead decreases the measure: if $a \in \{\mathsf{ren}, \mathsf{sea}_1\}$ then $\|s'\| < \|s\|$.*
- *Strong phase does not increase the measure: if $a \in \{\mathsf{sea}_2, \mathsf{sea}_3, \mathsf{gc}, \mathsf{sea}_4, \mathsf{sea}_5\}$ then $\|s'\| \leq \|s\|$.*

Proof. We proceed by cases on the transition a:

- Case β_λ: $e[x{\leftarrow}y\ z]\triangleleft K \rightsquigarrow_{\beta_v} e([x{\leftarrow}b]e'\{w{\leftarrow}z\})\triangleleft K$ where $e_K(y) = v$ and $v^\alpha = \lambda w.[\star{\leftarrow}b]e'$. First of all, note that by the size invariant:

$$\begin{array}{ll} & |[\star{\leftarrow}b]e'| \\ < & |\lambda w.[\star{\leftarrow}b]e'| \\ = & |v^\alpha| \\ =_{L.I.2.3} & |v| \\ \leq_{T.I.3} & \|s_0\| \end{array}$$

Then:

$$\begin{array}{ll} & \|e([x{\leftarrow}b]e'\{w{\leftarrow}z\}){\triangleleft}K\| \\ = & |e([x{\leftarrow}b]e'\{w{\leftarrow}z\})| + \|K\| \\ =_{L.I.2.1} & |e| + |[x{\leftarrow}b]e'\{w{\leftarrow}z\}| + \|K\| \\ =_{L.I.2.2} & |e| + |[x{\leftarrow}b]e'| + \|K\| \\ =_{L.I.2.4} & |e| + |[\star{\leftarrow}b]e'| + \|K\| \\ \leq & |e| + \|s_0\| + \|K\| \\ < & |e[x{\leftarrow}y\,z]| + \|K\| + \|s_0\| \\ = & \|e[x{\leftarrow}y\,z]{\triangleleft}K\| + \|s_0\| \end{array}$$

- Case β_i: $e[x{\leftarrow}y\,z] \triangleleft K \rightsquigarrow_{\beta_i} e[x{\leftarrow}b]e' \triangleleft K\langle\langle\cdot\rangle[w{\leftarrow}z]\rangle$ where $e_K(y) = v$ and $v^\alpha = \lambda w.[\star{\leftarrow}b]e'$. Then:

$$\begin{array}{ll} & \|e[x{\leftarrow}b]e'{\triangleleft}K\langle\langle\cdot\rangle[w{\leftarrow}z]\rangle\| \\ = & |e[x{\leftarrow}b]e'| + \|K\langle\langle\cdot\rangle[w{\leftarrow}z]\rangle\| \\ =_{L.I.5.2} & |e[x{\leftarrow}b]e'| + \|K\| \\ =_{L.I.2.1} & |e| + |[x{\leftarrow}b]e'| + \|K\| \\ =_{L.I.2.4} & |e| + |[\star{\leftarrow}b]e'| + \|K\| \\ \leq & |e| + \|s_0\| + \|K\| \\ < & |e[x{\leftarrow}y\,z]| + \|K\| + \|s_0\| \\ = & \|e[x{\leftarrow}y\,z]{\triangleleft}K\| + \|s_0\| \end{array}$$

- Case $\rightsquigarrow_{\mathsf{ren}}$: $e[x{\leftarrow}y]{\triangleleft}K \rightsquigarrow_{\mathsf{ren}} e\{x{\leftarrow}y\}{\triangleleft}K$ where $x \neq \star$. Then

$$\begin{array}{ll} & |e\{x{\leftarrow}y\}{\triangleleft}K| \\ = & |e\{x{\leftarrow}y\}| + \|K\| \\ =_{L.I.2.2} & |e| + \|K\| \\ < & |e[x{\leftarrow}y]| + \|K\| \\ = & \|e[x{\leftarrow}y]{\triangleleft}K\| \end{array}$$

- Case sea_1: $e[x{\leftarrow}b]{\triangleleft}K \rightsquigarrow_{\mathsf{sea}_1} e{\triangleleft}K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle$. Then

$$\begin{array}{ll} & \|e{\triangleleft}K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle\| \\ = & |e| + \|K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle\| \\ =_{L.I.5.2} & |e| + \|K\| + |b| \\ < & 1 + |e| + |b| + \|K\| \\ = & |e[x{\leftarrow}b]| + \|K\| \\ = & \|e[x{\leftarrow}b]{\triangleleft}K\| \end{array}$$

- Case sea_2: ${\triangleleft}K \rightsquigarrow_{\mathsf{sea}_2} {\triangleright}K$. Then

$$\begin{array}{ll} & \|{\triangleright}K\| \\ = & \|K\| \\ = & \|{\triangleleft}K\| \end{array}$$

- Case sea_3: $e{\triangleright}K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle \rightsquigarrow_{\mathsf{sea}_3} e[x{\leftarrow}b]{\triangleright}K$. Then

$$\begin{array}{ll} & \|e[x{\leftarrow}b]{\triangleright}K\| \\ = & \|K\| \\ < & \|K\| + |b| \\ =_{L.I.5.2} & \|K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle\| \\ = & \|e{\triangleright}K\langle\langle\cdot\rangle[x{\leftarrow}b]\rangle\| \end{array}$$

- Case gc: $e{\triangleright}K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle \rightsquigarrow_{\mathsf{gc}} e{\triangleright}K$. Then

$$\begin{array}{ll} & \|e{\triangleright}K\| \\ = & \|K\| \\ < & \|K\| + |v| \\ =_{L.I.5.2} & \|K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle\| \\ = & \|e{\triangleright}K\langle\langle\cdot\rangle[x{\leftarrow}v]\rangle\| \end{array}$$

- Case sea_4: $e\triangleright K\langle e'[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle \rightsquigarrow_{\mathsf{sea}_4} e'[x\leftarrow\lambda y.e]\triangleright K$. Then

$$\begin{array}{ll} & \|e'[x\leftarrow\lambda y.e]\triangleright K\| \\ = & \|K\| \\ = & \|K\| + \|e'[x\leftarrow\lambda y.\langle\cdot\rangle]\| \\ =_{L.I.5.1} & \|K\langle e'[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle\| \\ = & \|e\triangleright K\langle e'[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle\| \end{array}$$

- Case sea_5: $e\triangleright K\langle\langle\cdot\rangle[x\leftarrow\lambda y.e']\rangle \rightsquigarrow_{\mathsf{sea}_5} e'\triangleleft K\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle$. Then

$$\begin{array}{ll} & \|e'\triangleleft K\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle\| \\ = & |e'| + \|K\langle e[x\leftarrow\lambda y.\langle\cdot\rangle]\rangle\| \\ =_{L.I.5.1} & |e'| + \|K\| + \|e[x\leftarrow\lambda y.\langle\cdot\rangle]\| \\ = & |e'| + \|K\| \\ < & \|K\| + |\lambda y.e'| \\ =_{L.I.5.2} & \|K\langle\langle\cdot\rangle[x\leftarrow\lambda y.e']\rangle\| \\ = & \|e\triangleright K\langle\langle\cdot\rangle[x\leftarrow\lambda y.e']\rangle\| \end{array}$$

□

As a consequence of the two previous lemmas, we obtain the following combined bound on the number of machine transitions:

Corollary I.7 (Bi-linear number of overhead transitions)**.** *Let t be a λ-term and $\rho\colon t^\circ \rightsquigarrow^*_{SCAM} s$ a SCAM execution. Then $|\rho| \in \mathcal{O}((1+|\rho|_\beta)\cdot|t|)$.*

See p. 13 Cor. XI.5

Proof. Let $s_0 \coloneqq \underline{t}\triangleleft\langle\cdot\rangle$. Then $\|s_0\| = |\underline{t}|$, and by Lemma XI.1 $\|s_0\| \in \mathcal{O}(|t|)$. To prove the required statement, it suffices to show that overhead transitions are bilinear, *i.e.* that:

$$|\rho|_{\mathsf{ren}} + |\rho|_{\mathsf{sea}_1} + |\rho|_{\mathsf{sea}_2} + |\rho|_{\mathsf{sea}_3} + |\rho|_{\mathsf{gc}} + |\rho|_{\mathsf{sea}_4} + |\rho|_{\mathsf{sea}_5} \in \mathcal{O}((1+|\rho|_\beta)\cdot|t|).$$

Note that Lemma I.6 implies that $\|s\| \leq \|s_0\| + |\rho|_\beta \cdot |\underline{t}| - |\rho|_{\mathsf{ren}} - |\rho|_{\mathsf{sea}_1}$. Since $\|s_0\| = |\underline{t}|$ and $\|s\| \geq 0$, we obtain $|\rho|_{\mathsf{ren}} + |\rho|_{\mathsf{sea}_1} \leq (1+|\rho|_\beta)\cdot|\underline{t}|$, *i.e.* the number of ren and sea_1 transitions is bilinear. To bound the number of the remaining overhead transitions, just use Lemma I.4. □

B. Bi-linearity of the SCAM

In this subsection, we formalize the arguments in the paper about the cost of implementing SCAM execution on Random Access Machines. We proceed in the following way:

1) we prove in Lemma I.8 that all machine steps but garbage collection run in bi-linear time,
2) we derive in Cor. I.9 that the space consumption of the machine is also bi-linear,
3) we conclude noting that garbage collection, which runs in time proportional to the amount of space to be freed, is also bi-linear and therefore the SCAM runs in bi-linear time (Cor. I.10 which proves Thm. XI.6).

Lemma I.8 (The SCAM without garbage-collection is bilinear)**.** *For any λ-term t and any SCAM execution $\rho\colon t^\circ\triangleleft \rightsquigarrow^*_{SCAM} e\triangleright\langle\cdot\rangle$, the cost of implementing ρ on a RAM, excluding the cost of $\rightsquigarrow_{\mathsf{gc}}$ steps, is $\mathcal{O}((1+|\rho|_\beta)|t|)$.*

Proof. Consider that:

- each $\rightarrow_{\mathsf{x}f}$ step costs $\mathcal{O}(|t|)$ because, by Thm. I.3, the actual representation that encodes the value to be copied and renamed has size $\mathcal{O}(|t|)$. There are $|\rho|_\beta$ such steps.
- each overhead step except $\rightsquigarrow_{\mathsf{gc}}$ costs $\mathcal{O}(1)$ and there are $\mathcal{O}((1+|\rho|_\beta)|t|)$ such steps by Corollary XI.5.

Adding all the costs together yields the expected bound. □

Corollary I.9 (Cost of $\rightsquigarrow_{\mathsf{gc}}$ steps)**.** *A RAM in $\mathcal{O}((1+|\rho|_\beta)|t|)$ cannot create more than $\mathcal{O}((1+|\rho|_\beta)|t|)$ new constructors. Since the number of initial node is also $\mathcal{O}(|t|)$, garbage-collection cannot recover more than $\mathcal{O}((1+|\rho|_\beta)|t|)$ constructors and thus the overall cost of garbage-collection is also $\mathcal{O}((1+|\rho|_\beta)|t|)$.*

Corollary I.10 (The SCAM is bilinear)**.** *Let t be a λ-term and $\rho\colon t^\circ \rightsquigarrow^*_{SCAM} s$ a SCAM execution. Then ρ can be implemented on a RAM in $\mathcal{O}((1+|\rho|_\beta)|t|)$ time.*

See p. 13 Thm. XI.6

Proof. Immediate by Lemma I.8 and Corollary I.9. □

APPENDIX J
IMPLEMENTATION IN OCAML

We describe now an implementation of the SCAM in OCaml, that can be downloaded at https://tinyurl.com/y5remxo8. The code is meant to demonstrate concretely that every machine transition can be implemented in $\mathcal{O}(1)$. Moreover we tried very hard to minimize the memory footprint, avoiding doubly-linked data structures almost everywhere.

The implementation is also useful to experiment with ideas and variations and to trace execution on interesting terms. The code is not meant instead to be tested via benchmarks, to compare the implementation with other ones: it heavily employs mutation, which is costly in OCaml because of the write barrier of the garbage collector. Moreover OCaml is garbage collected, but our machine already takes care of garbage collecting. Therefore, for the sake of comparing with other implementations, the code should be rewritten in a low-level language like C.

The code is attached to the submission. It is self-contained and it implements a parser for pure terms, the crumbling translation, and the SCAM itself. It is also possible to compile the code to JavaScript and run it in a browser. In this case the user can write down a term to be reduced in the browser and look at every intermediate machine states.

The code snippets that we show in this section have been obtained by removing from the submitted code the pretty-printing statements used only to show how the machine runs.

A. Preliminaries: machine moves and zippers

The SCAM works on a graph of memory cells that encodes machine states. It crawls the graph looking for the next redex, which is reduced performing local graph modifications only. To reach the next redex, the machine moves in two different directions:

1) *bi-directional horizontal visits of environments*: the machine alternates left to right (in the open phase) and right to left (in the strong phase) walks on environments. Note that, because of sharing, environments are not simply lists but DAG structures. ‘
2) *bi-directional vertical visits of abstractions*: switching from an open phase to a strong one is done when entering into the body of an abstraction, itself an environment, and the opposite switch requires to exit the abstraction.

As we recall next, the space-conscious way of visiting bidirectionally a list is using a *zipper*, rather than doubly linking the list—the technique also smoothly scales up to trees. Our implementation uses the same principle, except that we have to deal with a rich graph structure (an enriched DAG), not just lists or trees.

1) Zippers: In functional programming, imperative data structures equipped with a cursor, such that the only modifications can happen locally where the cursor is placed, are effectively replaced by zipper-like data structures [42].

The zipper — the first zipper-like data structure ever discovered — represents a list and a cursor into it with two lists — actually two stacks: the first one is the tail of the list, i.e. the suffix of the list that starts at the cursor; the second one is the prefix of the list already traversed, with all pointers reversed to obtain again a list. Moving the cursor one position to the right or to the left, for example, just corresponds to popping the head of one list and pushing it on top of the other. Inserting an element where the cursor is corresponds to pushing an element on top of the suffix list, etc.

The zipper is interesting also in an imperative setting because it provides in $\mathcal{O}(1)$ most operations normally implemented using bi-directional lists, but it uses the same amount of space of a normal list (plus one additional pointer).

Zipper-like data structures can be obtained for all kind of algebraic data types via formal derivation [47]. In particular, it is easy to obtain a zipper-like data structure for trees. To obtain the zipper the idea is simply: every time a pointer is traversed, it is also reversed and the entry-point into the data structure becomes a tuples of forward pointers — to proceed in the visit — and backward pointers — to go back.

B. Data structures

We discuss now all the data structures used for the implementation, shown in Table I.

1) Bites and crumbled environments: An example of the representation in memory of a crumbled environment is given in Figure 3.

The type of bites is `bite`. A bite is either an occurrence of a variable, an abstraction or an application.

Free variables and variables bound by abstractions are encoded by `Var` cells. A `Var` holds a name (an integer), used only for pretty-printing purposes. Variables bound in the context are encoded by `Shared` cells, that hold (mutable) pointers to elements of the `exp_subst` datatype, which encodes explicit substitutions. We shall come back soon to the details of `exp_subst`.

Application cells `App` just hold two pointers to the two arguments.

Abstraction cells `Lam` hold a pointer to the bound variables and a (mutable) value of type `body_or_container`, that encodes the two possible forms of an abstraction in our zipper-like data structure: when the body of the abstraction is not being visited, the abstraction holds a pointer to its body, which is an environment encoded as a pointer to its rightmost explicit substitution. When the body is being visited, instead, the abstraction points back to its innermost enclosing abstraction, if it exists. To identify the enclosing abstraction, however, a single pointer to the bite is not sufficient to resume the visit later;

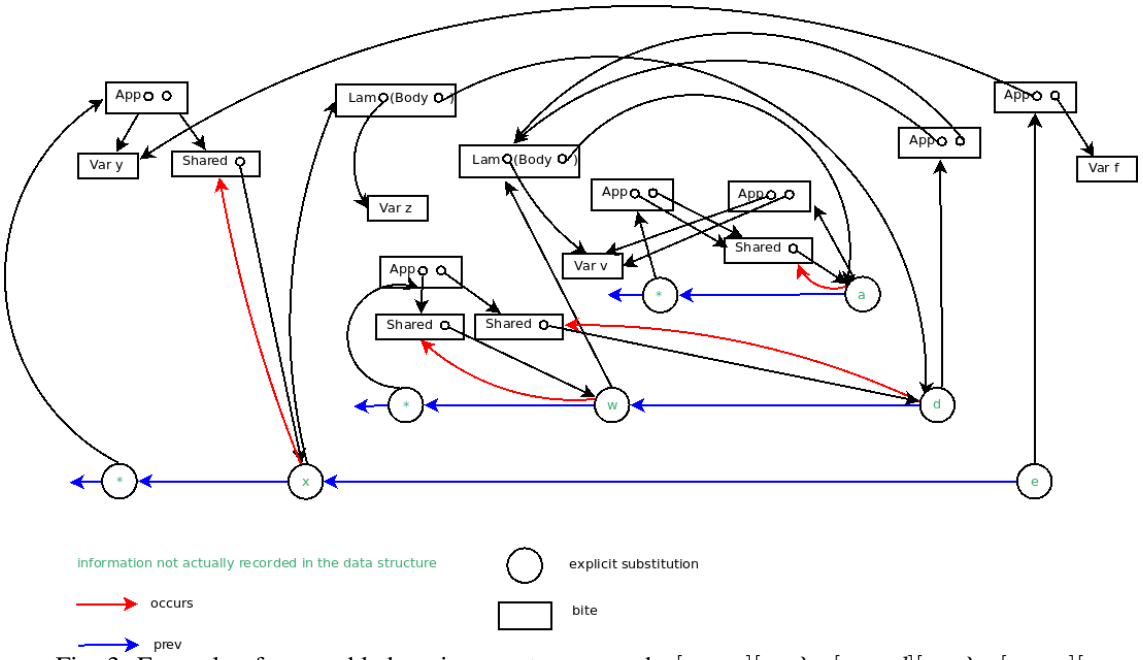

Fig. 3: Example of a crumbled environment as a graph: $[\star\leftarrow yx][x\leftarrow\lambda z.[\star\leftarrow wd][w\leftarrow\lambda v.[\star\leftarrow aa][a\leftarrow vv]][d\leftarrow zz]][e\leftarrow yf]$

```
type var = { name : int }

type exp_subst =
 { mutable content : bite
 ; mutable copying : bool
 ; mutable prev : exp_subst option
 ; mutable rc : int
 ; mutable occurs : bite option }
and environment = exp_subst
and revenvironment = exp_subst
and body_or_container =
 | Body of environment
 | Container of zipper option
and bite =
 | Var of var
 | Lam of {v: var ; mutable b: body_or_container}
 | App of bite * bite
 | Shared of {mutable c: exp_subst}
and zipper = environment option * revenvironment option
```

TABLE I: Data structures

instead we identify the enclosing abstraction — which is always the definiens of an explicit substitution — by pointing to its explicit substitution via a zipper over the environment the explicit substitution belongs to. We use the `Body` constructor to label the pointer to the body and the `Container` constructor to label the optional label to the zipper pointing to the enclosing abstraction.

The datatype `zipper` represents standard zippers over environments, which are lists. It is defined as a pair made of an environment, of type `environment`, and an environment where all pointers are reversed, of type `revenvironment`. Both environments are identified by the first explicit substitution of the list.

Explicit substitutions are represented by records of type `exp_subst` made of several mutable fields:

- `content`, a bite, it is the bite b of the ES $[x\leftarrow b]$. The variable x, instead, is unnamed and identified with the memory location of the cell.[5]
- `copying` is a boolean that is used to implement the linear graph copying algorithm described in [5], used to implement the α-renaming in the β-transitions of the SCAM. The algorithm is the same used in mark&sweep garbage collectors to

[5]We write (in green) the name x in the example graphs of this section for legibility purposes only, even if the name is lost in the implementation.

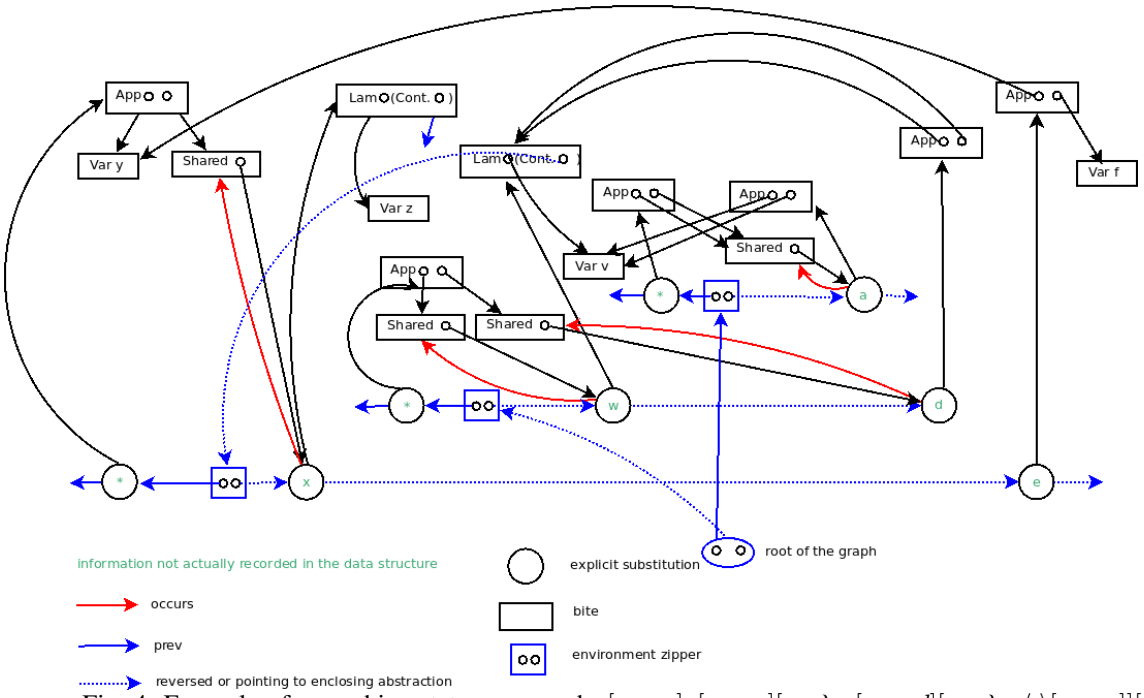

Fig. 4: Example of a machine state as a graph: $[\star\leftarrow aa]\triangleright[\star\leftarrow yx][x\leftarrow\lambda z.[\star\leftarrow wd][w\leftarrow\lambda v.\langle\cdot\rangle[a\leftarrow vv]][d\leftarrow zz]][e\leftarrow yf]$

copy a graph of linked cells to a new memory region without loosing sharing. Normally, the boolean is set to false. It temporarily becomes `true` during the copy when the cell has already been copied and therefore pointers to the cell are to be forwarded to the copy.

- `prev`, another optional explicit substitution, is the next substitution in the environment containing the explicit substitution. Therefore an environment is represented like in C lists by cells pointing to the next cell or to `None` if the cell is the last one of the list.
- `rc`, for *reference counter*, an integer, holding the number of occurrences in the graph of the variable bound by the explicit substitution. When it becomes zero the explicit substitution can be garbage collected.
- `occurs`, an optional bite: roughly, it indicates whether the variable bound by the ES has been introduced by the crumbling transformation. More precisely, variables generated during crumbling occurs exactly once; later, when rule $\rightsquigarrow_{\mathsf{ren}}$ is fired, a machine invariant (see Lemma F.1) grants the variable to be still occurring at most once. The `occurs` field, if set and if the explicit substitution is part of a pristine environment, points to the unique occurrence and it is used to implement rule $\rightsquigarrow_{\mathsf{ren}}$ in $\mathcal{O}(1)$. This is the only pointer that violates acyclicity of the memory cells graph.

2) Contexts and states: We recall here the definition of machine contexts and machine states:

$$\begin{array}{rrcl} \textsc{Machine contexts} & K & ::= & \langle\cdot\rangle e' \mid e[x\leftarrow\lambda y.K]e' \\ \textsc{State} & s & ::= & e \triangleleft K \mid e \triangleright K \end{array}$$

A machine state is of the form $e \bowtie K$ where e is a crumbled environment, K is a machine context and together they identify a precise position in the crumbled environment $K\langle e\rangle$. Observe that $K\langle e\rangle$ can be uniquely rewritten as $K\langle ee'\rangle$ where K is either $\langle\cdot\rangle$ or it has the form $K'\langle e_1[x\leftarrow\lambda y.\langle\cdot\rangle]e_2\rangle$. Therefore, to identify the position, it is sufficient to use a zipper for environments to encode ee' and a second optional zipper for environments to identify $[x\leftarrow\lambda y.\langle\cdot\rangle]$ inside $e_1[x\leftarrow\lambda y.\langle\cdot\rangle]e_2$. We already took care of identifying $e_1[x\leftarrow\lambda y.\langle\cdot\rangle]e_2$ inside $K'\langle e_1[x\leftarrow\lambda y.\langle\cdot\rangle]e_2\rangle$ by making the traversed abstraction $\lambda y.\langle\cdot\rangle$ point back, via another optional zipper, to its innermost enclosing abstraction.

Figure 4 shows the OCaml representation of a machine state $e\triangleright K$ such that $K\langle e\rangle$ is identical to the crumbled environment of Figure 3. The root of the graph is the pair of zippers just discussed. Dotted blue lines represents `prev` arcs that have been reversed or arcs used to make abstractions point to the innermost enclosing abstraction.

To summarize so far, the number of memory cells used to represent a machine state is linear in the number of cells used to represent a crumbled environment: for each traversed abstraction we added a new cell holding the two pointers of a zipper.

C. Machine moves and garbage collection

```
let rec gc {prev=p;content=c;_} =
 Option.iter gc p;
 gc_bite c
and gc_bite =
 function
    Var _ -> ()
  | Shared {c=v} -> v.rc <- v.rc - 1
  | App(t1,t2) -> gc_bite t1; gc_bite t2
  | Lam{b=Body e;_} -> gc e
  | Lam{b=Container _;_} -> assert false
```

TABLE II: Garbage collection

This section ends with the code of the right-to-left and left-to-right evaluation phases. The code also depends on two additional functions: `gc`, shown in Table II, that is meant to garbage collect the term and `copy_env` that creates in linear time an α-renamed copy of an environment using a modified mark&sweep algorithm described in [5].

Both evaluation functions take in input the entry point of the graph, i.e. the pairs made of the zipper and the optional zipper previously discussed. The call to start evaluation on a crumbled environment `e` is `eval_RL (Some e,None) None` where `(Some e, None)` is the initial zipper on the unvisited environment `e` and the second `None` signifies that we never crossed an abstraction so far.

The two `eval` functions trivially implement the reduction rules of the machine. The only interesting observations are:

1) `eval_LR` and `eval_LR` are mutual functions as expected.
2) all recursive calls to one of the two eval functions are in tail position and therefore there is no inner cost in space to evaluate them.
3) the calls to the two auxiliary functions `gc` and `copy_env` are not in tail position. These functions, moreover, have complexity linear in the size of their input both in space and in time.
4) all other operations except the calls to `gc` and `copy_env` have constant complexity as expected: they just increment/decrement numbers, read or assign pointers, pattern-match and build algebraic data types, and test pointer equalities.

The `gc` function written in OCaml is a bit weird: since OCaml has automatic garbage collection, `gc` just needs to traverse the data structure to decrement all the reference counters. Then the implementation of the $\rightsquigarrow_{\mathsf{gc}}$ rule will not use in the recursive call the explicit substitution to be garbage collected, triggering the garbage collector of OCaml. An implementation in C would just free all cells during the recursion.

```
let rec eval_RL ((n,z) as zip : zipper) (k : zipper option) =
 match n with
   None ->
    (* ⇝sea2 *)
    eval_LR (None,z) k
 | Some n ->
   match n.content with
    | (App(_ ,App _) | App(App _,_)
      |App(_, Lam _) | App(Lam _,_)
      |App(Shared {c={content=Lam{b=Container _;_};_}},_)) -> assert false
    | App(Shared {c={content=Lam{v=y;b=Body e;_};_} as r}, (Shared {c={content=Lam _;_} as y'})
      ) ->
       (* ⇝βv *)
       r.rc <- r.rc - 1 ;
       y'.rc <- y'.rc - 1 ;
       let e' = copy_env y y' n e in
       eval_RL (Some e',z) k
    | App(Shared {c={content=Lam{v=y;b=Body e};_} as r}, t) ->
       (* ⇝βi *)
       r.rc <- r.rc - 1 ;
       let y' = mk_exp_subst t in
       let e' = copy_env y y' n e in
       y'.prev <- z;
       eval_RL (Some e',Some y') k
    | Shared {c={content=(Shared _ | Var _);_} as c} when n.prev <> None ->
       (* ⇝ren *)
       (match n.occurs with
           Some (Shared r as o) ->
            c.occurs <- Some o;
```

```
            r.c <- c ;
            eval_RL (n.prev,z) k
          | _ -> assert false) ;
    | (Lam _ | Var _ | App(Var _, _)
       |Shared _ | App(Shared _, _)) ->
         (* ⇝sea1 *)
        let p = n.prev in
       eval_RL (Some e',z) k
    | App(Shared {c={content=Lam{v=y;b=Body e};_} as r}, t) ->
       (* ⇝βi *)
       r.rc <- r.rc - 1 ;
       let y' = mk_exp_subst t in
       let e' = copy_env y y' n e in
       y'.prev <- z;
       eval_RL (Some e',Some y') k
    | Shared {c={content=(Shared _ | Var _);_} as c} when n.prev <> None ->
        (* ⇝ren *)
        (match n.occurs with
            Some (Shared r as o) ->
             c.occurs <- Some o;
             r.c <- c ;
             eval_RL (n.prev,z) k
          | _ -> assert false) ;
    | (Lam _ | Var _ | App(Var _, _)
       |Shared _ | App(Shared _, _)) ->
         (* ⇝sea1 *)
        let p = n.prev in
        n.prev <- z;
        eval_RL (p,Some n) k
and eval_LR ((n,z) as zip : zipper) (k : zipper option) =
   match z,k with
      None,None ->
       (* normal form reached! *)
       n
   | None,Some (n',Some ({prev=z'';content=Lam({b=Container k';_} as r);_} as z')) ->
      (* ⇝sea4 *)
      r.b <- Body (match n with None -> assert false | Some n -> n);
      z'.prev <- n';
      eval_LR (Some z',z'') k'
   | None,Some _ -> assert false
   | Some ({prev=p;rc;_} as zz),_ ->
      match zz.content with
       | Lam {b=Body e;_} when rc = 0 ->
           (* ⇝gc *)
          gc e;
          eval_LR (n,p) k
       | Lam ({b=Body e;_} as r) ->
           (* ⇝sea5 *)
          r.b <- Container k;
          eval_RL (Some e,None) (Some (n,z))
       | Lam {b=Container _;_} ->
          assert false
       | (Var _ | Shared _ | App _) ->
           (* ⇝sea3 *)
          zz.prev <- n;
          eval_LR (Some zz,p) k
```

APPENDIX K
PROOFS OF SECTION XII (IMPLOSIVENESS AT WORK)

Proposition K.1 (Implosive family)**.** *Let t_n and u_n as in Section XII.*

1) External strategy (exponentially many steps)*: $t_n(\to_{\mathsf{xm}}\to_{\mathsf{xe}})^{2^n-1}u_n$ and u_n is a strong fireball.*

2) SCAM Implosion (linearly many steps)*: $\rho_n : t_n^\circ \rightsquigarrow^*_{SCAM} s_n$ with $s_n{\downarrow} = u_n$ and $|\rho_n|_\beta = n$.*

See p. 13 Prop. XII.1

Proof.

1) By induction on n.

- Case $n = 1$:
$$t_1 = \pi I = (\lambda x.\lambda y.((yx)x))I \to_{\mathsf{xm}} \lambda y.((yx)x)[x{\leftarrow}I] \to_{\mathsf{xe}} \lambda y.((yI)I) = u_1$$
and u_1 is evidently a strong fireball.
- Case $n + 1$:
$$t_{n+1} = \pi(\lambda z.t_n) = (\lambda x.\lambda y.((yx)x))(\lambda z.t_n) \to_{\mathsf{xm}} (\lambda y.((yx)x))[x{\leftarrow}\lambda z.t_n] \to_{\mathsf{xe}} \lambda y.((y(\lambda z.t_n))(\lambda z.t_n)).$$
By *i.h.*, $t_n(\to_{\mathsf{xm}}\to_{\mathsf{xe}})^{2^n-1}u_n$ and the two copies of t_n in $\lambda y.((y(\lambda z.t_n))(\lambda z.t_n))$ are inside an external evaluation context. Then $\lambda y.((y(\lambda z.t_n))(\lambda z.t_n))(\to_{\mathsf{xm}}\to_{\mathsf{xe}})^{2^n-1}\lambda y.((y(\lambda z.u_n))(\lambda z.t_n))(\to_{\mathsf{xm}}\to_{\mathsf{xe}})^{2^n-1}\lambda y.((y(\lambda z.u_n))(\lambda z.u_n)) = u_{n+1}$. The number of $\to_{\mathsf{xm}}\to_{\mathsf{xe}}$ double steps is $2^n - 1 + 2^n - 1 + 1 = 2^{n+1} - 1$. By *i.h.*, u_n is a strong fireball and so u_{n+1} is a strong fireball.

2) We sketch the idea. The crumbling of t_{n+1} is:
$$[\star{\leftarrow}wz'][w{\leftarrow}\lambda x.\underline{\lambda y.((yx)x)}][z'{\leftarrow}\lambda z.\underline{t_n}]$$
Thus the machine does
$$\begin{array}{ll}
[\star{\leftarrow}wz'][w{\leftarrow}\lambda x.\underline{\lambda y.((yx)x)}][z'{\leftarrow}\lambda z.\underline{t_n}]\triangleleft\langle\cdot\rangle & \rightsquigarrow_{\mathsf{sea}_1}\\
[\star{\leftarrow}wz'][w{\leftarrow}\lambda x.\underline{\lambda y.((yx)x)}]\triangleleft\langle\cdot\rangle[z'{\leftarrow}\lambda z.\underline{t_n}] & \rightsquigarrow_{\mathsf{sea}_1}\\
[\star{\leftarrow}wz']\triangleleft\langle\cdot\rangle[w{\leftarrow}\lambda x.\underline{\lambda y.((yx)x)}][z'{\leftarrow}\lambda z.\underline{t_n}] & \rightsquigarrow_{\beta_v}\\
[\star{\leftarrow}\lambda y'.\underline{((y'z')z')}]\triangleleft\langle\cdot\rangle[w{\leftarrow}\lambda x.\underline{\lambda y.((yx)x)}][z'{\leftarrow}\lambda z.\underline{t_n}] & \rightsquigarrow_{\mathsf{sea}_1}\\
\triangleleft\langle\cdot\rangle[\star{\leftarrow}\lambda y'.\underline{((y'z')z')}][w{\leftarrow}\lambda x.\underline{\lambda y.((yx)x)}][z'{\leftarrow}\lambda z.\underline{t_n}] &
\end{array}$$
At this point the machine changes phase and enters $\lambda y'$. Since the body is normal and the occurrences of z' appear as arguments, the machine shall go through it without performing any β-transition, and thus without copying $\lambda z.\underline{t_n}$. Thus the machine gets to the state
$$[\star{\leftarrow}\lambda y'.\underline{((y'z')z')}]\triangleright\langle\cdot\rangle[w{\leftarrow}\lambda x.\underline{\lambda y.((yx)x)}][z'{\leftarrow}\lambda z.\underline{t_n}]$$
Next, the machine garbage collects the ES on w and enters λz.
$$\begin{array}{ll}
\rightsquigarrow_{\mathsf{gc}} & [\star{\leftarrow}\lambda y'.\underline{((y'z')z')}]\triangleright\langle\cdot\rangle[z'{\leftarrow}\lambda z.\underline{t_n}]\\
\rightsquigarrow_{\mathsf{sea}_5} & \underline{t_n}\triangleleft[\star{\leftarrow}\lambda y'.\underline{((y'z')z')}][z'{\leftarrow}\lambda z.\langle\cdot\rangle]
\end{array}$$
Note that t_n is closed, so that its execution does not depend on the context $[\star{\leftarrow}\lambda y'.\underline{((y'z')z')}][z'{\leftarrow}\lambda z.\langle\cdot\rangle]$. Therefore the machine repeats the same sequence of transitions—that contains only one β-transition—for t_n. Therefore, the machine executes t_{n+1} doing only $n + 1$ β-transitions. By bilinearity, the whole execution has length $\mathcal{O}(n + 1)$. □