Distilling Abstract Machines

Beniamino Accattoli
Carnegie Mellon University & Università di Bologna
beniamino.accattoli@gmail.com

Pablo Barenbaum
University of Buenos Aires – CONICET
pbarenbaum@dc.uba.ar

Damiano Mazza
CNRS, UMR 7030, LIPN, Université Paris 13, Sorbonne Paris Cité
Damiano.Mazza@lipn.univ-paris13.fr

Abstract

It is well-known that many environment-based abstract machines can be seen as strategies in lambda calculi with explicit substitutions (ES). Recently, graphical syntaxes and linear logic led to the linear substitution calculus (LSC), a new approach to ES that is halfway between big-step calculi and traditional calculi with ES. This paper studies the relationship between the LSC and environment-based abstract machines. While traditional calculi with ES simulate abstract machines, the LSC rather distills them: some transitions are simulated while others vanish, as they map to a notion of structural congruence. The distillation process unveils that abstract machines in fact implement weak linear head reduction, a notion of evaluation having a central role in the theory of linear logic. We show that such a pattern applies uniformly in call-by-name, call-by-value, and call-by-need, catching many machines in the literature. We start by distilling the KAM, the CEK, and the ZINC, and then provide simplified versions of the SECD, the lazy KAM, and Sestoft's machine. Along the way we also introduce some new machines with global environments. Moreover, we show that distillation preserves the time complexity of the executions, i.e. the LSC is a complexity-preserving abstraction of abstract machines.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords Abstract machines, explicit substitutions, linear logic.

1. Introduction

In the theory of higher-order programming languages, abstract machines and explicit substitutions are two tools used to model the execution of programs on real machines while omitting many details of the actual implementation. Abstract machines can usually be seen as evaluation strategies in calculi of explicit substitutions (see at least [12, 15, 25, 31]), that can in turn be interpreted as small-step cut-elimination strategies in sequent calculi [10].

Another tool providing a fine analysis of higher-order evaluation is linear logic, especially via the new perspectives on cut-elimination provided by *proof nets*, its graphical syntax. Explicit substitutions (ES) have been connected to linear logic by Kesner and co-authors in a sequence of works [21, 27, 28], culminating in the *linear substitution calculus* (LSC), a new formalism with ES behaviorally isomorphic to proof nets (introduced in [4], developed in [1–3, 5, 6], and bearing similarities with calculi by De Bruijn [20], Nederpelt [37], and Milner [36]). Since linear logic can model all evaluation schemes (call-by-name/value/need) [34], the LSC can express them modularly, by minor variations on rewriting rules and evaluation contexts. In this paper we revisit the relationship between environment-based abstract machines and ES. Traditionally, ES simulate machines. The LSC, instead, distills them.

Traditional vs Contextual ES. Traditional calculi with ES (see [26] for a survey) implement β-reduction $(\lambda x.t)u \to_\beta t\{x{\leftarrow}u\}$ introducing an annotation (the explicit substitution $[x{\leftarrow}u]$),

$$(\lambda x.t)u \quad \to_{\mathtt{B}} \quad t[x{\leftarrow}u]$$

and percolating it through the term structure,

$$\begin{array}{rcl} (tw)[x{\leftarrow}u] & \to_{@} & t[x{\leftarrow}u]w[x{\leftarrow}u] \\ (\lambda x.t)[y{\leftarrow}u] & \to_{\lambda} & \lambda x.t[y{\leftarrow}u] \end{array} \tag{1}$$

until they reach variable occurrences on which they finally substitute or get garbage collected,

$$\begin{array}{rcl} x[x{\leftarrow}u] & \to_{\mathtt{var}} & u \\ y[x{\leftarrow}u] & \to_{\neq} & y \end{array}$$

The LSC, instead, is based on a *contextual* view of evaluation and substitution, also known as *at a distance*. The idea is that one can get rid of the rules percolating through the term structure — *i.e.* @ and λ — by introducing contexts C (i.e. terms with a hole $\langle\cdot\rangle$) and generalizing the base cases, obtaining just two rules, *linear substitution* (ls) and *garbage collection* (gc):

$$\begin{array}{rcll} C\langle x\rangle[x{\leftarrow}u] & \to_{\mathtt{ls}} & C\langle u\rangle & \\ t[x{\leftarrow}u] & \to_{\mathtt{gc}} & t & \text{if } x \notin \mathtt{fv}(t) \end{array}$$

Dually, the rule creating substitutions (B) is generalized to act up to a context of substitutions $[\ldots{\leftarrow}\ldots] := [x_1{\leftarrow}w_1]\ldots[x_k{\leftarrow}w_k]$ obtaining rule dB (B at a distance):

$$(\lambda x.t)[\ldots{\leftarrow}\ldots]u \quad \to_{\mathtt{dB}} \quad t[x{\leftarrow}u][\ldots{\leftarrow}\ldots]$$

Logical Perspective on the LSC. From a sequent calculus point of view, rules @ and λ, corresponding to *commutative* cut-elimination cases, are removed and integrated — via the use of contexts — directly in the definition of the *principal* cases B, var and $\neq$, obtaining the contextual rules dB, ls, and gc. This is the term analogous of the removal of commutative cases provided by proof nets. From a linear logic point of view, $\to_{\mathtt{dB}}$ can be identified with the multiplicative cut-elimination case $\to_{\mathtt{m}}$, while $\to_{\mathtt{ls}}$ and $\to_{\mathtt{gc}}$ correspond to exponential cut-elimination. Actually, garbage collection has a special status, as it can always be postponed. We will then identify exponential cut-elimination $\to_{\mathtt{e}}$ with linear substitution $\to_{\mathtt{ls}}$ alone.

ICFP 2014, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

The LSC has a simple meta-theory, and is halfway between traditional calculi with ES — with whom it shares the small-step dynamics — and λ-calculus — of which it retains most of the simplicity.

Distilling Abstract Machines. Abstract machines implement the traditional approach to ES, by

1. *Weak Evaluation*: forbidding reduction under abstraction (no rule $\rightarrow_\lambda$ in (1)),
2. *Evaluation Strategy*: looking for redexes according to some notion of weak evaluation context E,
3. *Context Representation*: using environments e (aka lists of substitutions) and stacks π (lists of terms) to keep track of the current evaluation context.

The LSC distills — *i.e.* factorizes — abstract machines. The idea is that one can represent the strategy of an abstract machine by directly plugging the evaluation context in the contextual substitution/exponential rule, obtaining:

$$E\langle x\rangle[x{\leftarrow}u] \quad \overset{E}{\rightarrow}_{\mathtt{e}} \quad E\langle u\rangle$$

and factoring out part of the machine that just looks for the next redex to reduce. By defining $\multimap$ as the closure of $\overset{E}{\rightarrow}_{\mathtt{e}}$ and $\rightarrow_{\mathtt{m}}$ by evaluation contexts E, one gets a clean representation of the machine strategy.

The mismatch between the two approaches is in rule $\rightarrow_{@}$, that contextually — by nature — cannot be captured. In order to get out of this *cul-de-sac*, the very idea of *simulation* of an abstract machine must be refined to that of *distillation*.

The crucial observation is that the equivalence $\equiv$ induced by $\rightarrow_{@} \cup \rightarrow_{\mathtt{gc}}$ has the same special status of $\rightarrow_{\mathtt{gc}}$, *i.e.* it can be postponed without affecting reduction lengths. More abstractly, $\equiv$ is a *strong bisimulation* with respect to $\multimap$, *i.e.* it verifies (note *one* step to *one* step, and *viceversa*)

$$\begin{array}{ccc} t & \multimap & r \\ \equiv & & \equiv \\ u & \dashrightarrow & q \end{array} \qquad\qquad \begin{array}{ccc} t & \dashrightarrow & r \\ \equiv & & \equiv \\ u & \multimap & q \end{array}$$

Now, $\equiv$ can be considered as a *structural equivalence* on the language. Indeed, the strong bisimulation property states that the transformation expressed by $\equiv$ is irrelevant with respect to $\multimap$, in particular $\equiv$-equivalent terms have $\multimap$-evaluations of the same length ending in $\equiv$-equivalent terms (and this holds even locally).

Abstract machines then are *distilled*: the logically relevant part of the substitution process is retained by $\multimap$ while both the search of the redex $\rightarrow_{@}$ and garbage collection $\rightarrow_{\neq}$ are isolated into the equivalence $\equiv$. Essentially, $\multimap$ captures principal cases of cut-elimination while $\equiv$ encapsulate the commutative ones (plus garbage collection, corresponding to principal cut-elimination involving weakenings).

Case Studies. We will analyze along these lines many abstract machines. Some are standard (KAM [29], CEK [23], ZINC [32]), some are new (MAM, WAM), and of others we provide simpler versions (SECD [30], Lazy KAM [15, 19], Sestoft's [39]). The previous explanation is a sketch of the distillation of the KAM, but the approach applies *mutatis mutandis* to all the other machines, encompassing most realizations of call-by-name, call-by-value, and call-by need evaluation. The main contribution of the paper is indeed a modular *contextual* theory of abstract machines. We start by distilling some standard cases, and then rationally reconstruct and simplify non-trivial machines as the SECD, the lazy KAM, and Sestoft's abstract machine for call-by-need (deemed SAM), by enlightening their mechanisms as different encoding of evaluation contexts, modularly represented in the LSC.

Call-by-Need. Along the way, we show that the contextual (or *at a distance*) approach of the LSC naturally leads to simple machines with just one global environment, as the newly introduced MAM (M for Milner). Such a feature is then showed to be a key ingredient of call-by-need machines, by using it to introduce a new and simple call-by-need machine, the WAM (W for Wadsworth), and then showing how to obtain (simplifications of) the Lazy KAM and the SAM by simple tweaks.

Distillation is Complexity-Preserving. It is natural to wonder what is lost in the distillation process. What is the asymptotic impact of distilling machine executions into $\multimap$? Does it affect in any way the complexity of evaluation? We will show that *nothing is lost*, as machine executions are only linearly longer than $\multimap$. More precisely, they are *bilinear*, *i.e.* they are linear in 1) the length of $\multimap$, and in 2) the size $|t|$ of the starting term t. In other words, the search of redexes and garbage collection can be safely ignored in quantitative (time) analyses, *i.e.* the LSC and $\multimap$ provide a complexity-preserving abstraction of abstract machines. While in call-by-name and call-by-value such an analysis follows from an easy local property of machine executions, the call-by-need case is subtler, as such a local property does not hold and bilinearity can be established only via a global analysis.

Linear Logic and Weak Linear Head Reduction. Beyond the contextual view, our work also unveils a deep connection between abstract machines and linear logic. The strategies modularly encoding the various machines (generically noted $\multimap$ and parametric in a fixed notion of evaluation contexts) are in fact call-by-name/value/need versions of *weak linear head reduction* (WLHR), a fundamental notion in the theory of linear logic [2, 14, 17, 22, 35]. This insight —- due to Danos and Regnier for the KAM [16] — is not ours, but we develop it in a simpler and tighter way, modularly lifting it to many other abstract machines.

Call-by-Name. The call-by-name case (catching the KAM and the new MAM) is in fact special, as our distillation theorem has three immediate corollaries, following from results about WLHR in the literature:

1. *Invariance*: it implies that the length of a KAM/MAM execution is an an invariant time cost model (*i.e.* polynomially related to, say, Turing machines, in both directions), given that in [3] the same is shown for WLHR.
2. *Evaluation as Communication*: we implicitly establish a link between the KAM/MAM and the π-calculus, given that the evaluation of a term via WLHR is isomorphic to evaluation via Milner's encoding in the π-calculus [2].
3. *Plotkin's Approach*: our study complements the recent [6], where it is shown that WLHR is a standard strategy of the LSC. The two works together provide the lifting to explicit substitutions of Plotkin's approach of relating a machine (the SECD machine in that case, the KAM/MAM in ours) and a calculus (the call-by-value λ-calculus and the LSC, respectively) via a standardization theorem and a standard strategy [38].

Related Work. Beyond the already cited works, Danvy and coauthors have studied abstract machines in a number of works and ways (see at least [7, 8, 11, 12, 19]). What here we call *commutative transitions* essentially corresponds to what Danvy and Nielsen call *decompose* phase in [18]. The call-by-need calculus we use—that is a contextual re-formulation of Maraist, Odersky, and Wadler's calculus [33]—is a novelty of this paper. It is simpler than both Ariola and Felleisen's [9] and Maraist, Odersky, and Wadler's calculi because it does not need any re-association axioms. Morally, it is a version with let-bindings (avatars of ES) of Chang and Felleisen's calculus [13]. A similar calculus is used by Danvy and Zerny in

[19], where, similarly to us, they explore various call-by-need machines in relation to various calculi. The differences are that 1) they focus on the contrast between store-based and storeless machines, 2) they do not stress the value of contexts as we do here, 3) they do not connect their study to linear logic. Another call-by-need machine, with whom we do not deal with, appears in [24].

2. Preliminaries on the Linear Substitution Calculus

Terms and Contexts. The language of the *weak linear substitution calculus* (WLSC) is generated by the following grammar:

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

The constructor $t[x{\leftarrow}u]$ is called an *explicit substitution* (of u for x in t). The usual (implicit) substitution is instead denoted by $t\{x{\leftarrow}u\}$. Both $\lambda x.t$ and $t[x{\leftarrow}u]$ bind x in t, with the usual notion of α-equivalence. Values, noted v, do not include variables: this is a standard choice in the study of abstract machines.

Contexts are terms with one occurrence of the hole $\langle\cdot\rangle$, an additional constant. We will use many different contexts. The most general ones will be *weak contexts* W (*i.e.* not under abstractions), which are defined by:

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

The *plugging* $W\langle t\rangle$ (resp. $W\langle W'\rangle$) of a term t (resp. context W') in a context W is defined as $\langle t\rangle := t$ (resp. $\langle W'\rangle := W'$), $(Wt)\langle W'\rangle := W\langle W'\rangle t$ (resp. $(Wu)\langle W'\rangle := W\langle W'\rangle u$), and so on. The set of free variables of a term t (or context W) is denoted by $\mathtt{fv}(t)$ (resp. $\mathtt{fv}(W)$). Plugging in a context may capture free variables (replacing holes on the left of substitutions). These notions will be silently extended to all the contexts used in the paper.

Rewriting Rules. On the above terms, one may define several variants of the LSC by considering two elementary rewriting rules, *distance-β* ($\mathtt{dB}$) and *linear substitution* ($\mathtt{ls}$), each one coming in two variants, call-by-name and call-by-value (the latter variants being abbreviated by $\mathtt{dBv}$ and $\mathtt{lsv}$), and pairing them in different ways and with respect to different evaluation contexts.

The rewriting rules rely in multiple ways on contexts. We start by defining *substitution contexts*, generated by

$$L \quad ::= \langle\cdot\rangle \mid L[x{\leftarrow}t].$$

A term of the form $L\langle v\rangle$ is an *answer*. Given a family of contexts C, the two variants of the elementary rewriting rules, also called *root rules*, are defined as follows:

$$\begin{array}{rcl} L\langle\lambda x.t\rangle u & \mapsto_{\mathtt{dB}} & L\langle t[x{\leftarrow}u]\rangle \\ L\langle\lambda x.t\rangle L'\langle v\rangle & \mapsto_{\mathtt{dBv}} & L\langle t[x{\leftarrow}L'\langle v\rangle]\rangle \\ C\langle x\rangle[x{\leftarrow}u] & \mapsto_{\mathtt{ls}} & C\langle u\rangle[x{\leftarrow}u] \\ C\langle x\rangle[x{\leftarrow}L\langle v\rangle] & \mapsto_{\mathtt{lsv}} & L\langle C\langle v\rangle[x{\leftarrow}v]\rangle \end{array}$$

In the linear substitution rules, we assume that $x \in \mathtt{fv}(C\langle x\rangle)$, *i.e.*, the context C does not capture the variable x, and we also silently work modulo α-equivalence to avoid variable capture in the rewriting rules. Moreover, we use the notations $\overset{C}{\mapsto}_{\mathtt{ls}}$ and $\overset{C}{\mapsto}_{\mathtt{lsv}}$ to specify the family of contexts used by the rules, with C being the meta-variable ranging over such contexts.

All of the above rules are *at a distance* (or *contextual*) because their definition involves contexts. Distance-β and linear substitution correspond, respectively, to the so-called *multiplicative* and *exponential* rules for cut-elimination in proof nets. The presence of contexts is how locality on proof nets is reflected on terms.

A linear substitution calculus is defined by a choice of root rules, *i.e.*, one of $\mathtt{dB}/\mathtt{dBv}$ and one of $\mathtt{ls}/\mathtt{lsv}$, and a family of *evaluation contexts*. The chosen distance-β (resp. linear substitution) root rule is generically denoted by $\mapsto_{\mathtt{m}}$ (resp. $\mapsto_{\mathtt{e}}$). If E ranges over a fixed notion of evaluation context, the context-closures of the root rules are denoted by $\multimap_{\mathtt{m}} := E\langle\mapsto_{\mathtt{m}}\rangle$ and $\multimap_{\mathtt{e}} := E\langle\mapsto_{\mathtt{e}}\rangle$, where $\mathtt{m}$ (resp. $\mathtt{e}$) stays for *multiplicative* (*exponential*). The rewriting relation defining the calculus is then $\multimap := \multimap_{\mathtt{m}} \cup \multimap_{\mathtt{e}}$.

Calculi. We consider four calculi, noted $\mathtt{Name}$, $\mathtt{Value}^{\mathtt{LR}}$, $\mathtt{Value}^{\mathtt{RL}}$, and $\mathtt{Need}$, and defined in the left half of Tab. 1. They correspond to four standard evaluation strategies for functional languages. We are actually slightly abusing the terminology, because — as we will show — they are *deterministic* calculi and thus should be considered as strategies. Our abuse is motivated by the fact that they are not strategies in the same calculus.

The evaluation contexts for $\mathtt{Name}$ are called *weak head contexts* and implement a strategy known as *weak linear head reduction*. The original presentation of this strategy does not use explicit substitutions [16, 35]. The presentation in use here has already appeared in [2, 6] (see also [1, 3]) as the weak head strategy of the *linear substitution calculus* (which is obtained by considering *all* contexts as evaluation contexts), and it avoids many technicalities of the original one. In particular, its relationship with the KAM is extremely natural, as we will show.

For call-by-value calculi, *left-to-right* ($\mathtt{Value}^{\mathtt{LR}}$) and *right-to-left* ($\mathtt{Value}^{\mathtt{RL}}$) refer to the evaluation order of applications, *i.e.* they correspond to *function body first* and *argument first*, respectively. The two calculi we consider here can be seen as strategies of a small-step variant of the *value substitution calculus*, the (big-step) call-by-value calculus at a distance introduced and studied in [5].

The call-by-need calculus $\mathtt{Need}$ is a novelty of this paper, and can be seen either as a version at a distance of the calculi of [9, 33] or as a version with explicit substitution of the one in [13]. It fully exploits the fact that the two variants of the root rules may be combined: the β-rule is call-by-name, which reflects the fact that, operationally, the strategy is *by name*, but substitution is call-by-value, which forces arguments to be evaluated before being substituted, reflecting the *by need* content of the strategy. Note that its evaluation contexts extends the weak head contexts for call-by-name with a clause $(N'\langle x\rangle[x{\leftarrow}N])$ turning them into *hereditarily weak head contexts*. This new clause is how sharing is implemented by the reduction strategy. The general (non-deterministic) calculus is obtained by closing the root rules by *all* contexts, but its study is omitted. What we deal with here can be thought as its standard strategy (stopping on a sort of weak head normal form).

As mentioned above, an essential property of all these four calculi is that they are deterministic, because they implement a reduction strategy.

Proposition 2.1 (Determinism)**.** *The reduction relations of the four calculi of Tab. 1 are deterministic: in each calculus, if E_1, E_2 are evaluation contexts and if r_1, r_2 are redexes (i.e., terms matching the left hand side of the root rules defining the calculus), $E_1\langle r_1\rangle = E_2\langle r_2\rangle$ implies $E_1 = E_2$ and $r_1 = r_2$, so that there is at most one way to reduce a term.*

Proof. See Sect. A in the appendix (page 13). □

Structural equivalence. Another common feature of the four calculi is that they come with a notion of *structural equivalence*, denoted by $\equiv$. Consider Fig. 1. For call-by-name and call-by-value calculi, $\equiv$ is defined as the smallest equivalence relation containing the closure by weak contexts of $=_\alpha \cup \equiv_{gc} \cup \equiv_{dup} \cup \equiv_{@} \cup \equiv_{com} \cup \equiv_{[\cdot]}$ where $=_\alpha$ is α-equivalence. Call-by-need evaluates inside some substitutions (but not any substitution) and thus axioms as $\equiv_{dup}$ and $\equiv_{@}$ are too strong. Therefore, the structural equivalence for call-by-need is the one generated by $\equiv_{@l} \cup \equiv_{com} \cup \equiv_{[\cdot]}$.

Structural equivalence represents the fact that certain manipulations on explicit substitutions are computationally irrelevant, in the

Calculus	Evaluation contexts	$\mapsto_{\mathtt{m}}$	$\mapsto_{\mathtt{e}}$	$\multimap_{\mathtt{m}}$	$\multimap_{\mathtt{e}}$
Name	$H ::= \langle\cdot\rangle \mid Ht \mid H[x{\leftarrow}t]$	$\mapsto_{\mathtt{dB}}$	$\overset{H}{\mapsto}_{\mathtt{ls}}$	$H\langle\mapsto_{\mathtt{dB}}\rangle$	$H\langle\overset{H}{\mapsto}_{\mathtt{ls}}\rangle$
Value$^{\mathtt{LR}}$	$V ::= \langle\cdot\rangle \mid Vt \mid L\langle v\rangle V \mid V[x{\leftarrow}t]$	$\mapsto_{\mathtt{dBv}}$	$\overset{V}{\mapsto}_{\mathtt{lsv}}$	$V\langle\mapsto_{\mathtt{dB}}\rangle$	$V\langle\overset{V}{\mapsto}_{\mathtt{ls}}\rangle$
Value$^{\mathtt{RL}}$	$S ::= \langle\cdot\rangle \mid SL\langle v\rangle \mid tS \mid S[x{\leftarrow}t]$	$\mapsto_{\mathtt{dBv}}$	$\overset{S}{\mapsto}_{\mathtt{lsv}}$	$S\langle\mapsto_{\mathtt{dB}}\rangle$	$S\langle\overset{S}{\mapsto}_{\mathtt{ls}}\rangle$
Need	$N ::= \langle\cdot\rangle \mid Nt \mid N[x{\leftarrow}t] \mid N'\langle x\rangle[x{\leftarrow}N]$	$\mapsto_{\mathtt{dB}}$	$\overset{N}{\mapsto}_{\mathtt{lsv}}$	$N\langle\mapsto_{\mathtt{dB}}\rangle$	$N\langle\overset{N}{\mapsto}_{\mathtt{ls}}\rangle$

Table 1. The four linear substitution calculi.

$$
\begin{array}{rcll|rcll}
t[x{\leftarrow}u] & \equiv_{gc} & t & \text{if } x \notin \mathtt{fv}(t) & t[x{\leftarrow}u] & \equiv_{dup} & t_{[y]_x}[x{\leftarrow}u][y{\leftarrow}u] & \\
t[x{\leftarrow}u][y{\leftarrow}w] & \equiv_{com} & t[y{\leftarrow}w][x{\leftarrow}u] & \text{if } y \notin \mathtt{fv}(u) & (tw)[x{\leftarrow}u] & \equiv_{@} & t[x{\leftarrow}u]w[x{\leftarrow}u] & \\
t[x{\leftarrow}u][y{\leftarrow}w] & \equiv_{[\cdot]} & t[x{\leftarrow}u[y{\leftarrow}w]] & \text{if } y \notin \mathtt{fv}(t) & (tw)[x{\leftarrow}u] & \equiv_{@l} & t[x{\leftarrow}u]w & \text{if } x \notin \mathtt{fv}(w)
\end{array}
$$

Figure 1. Axioms for structural equivalences. In $\equiv_{dup}$, $t_{[y]_x}$ denotes a term obtained from t by renaming some (possibly none) occurrences of x as y.

sense that they yield behaviorally equivalent terms. Technically, it is a *strong bisimulation*:

Proposition 2.2 ($\equiv$ is a Strong Bisimulation)**.** *Let $\multimap_{\mathtt{m}}$, $\multimap_{\mathtt{e}}$ and $\equiv$ be the reduction relations and the structural equivalence relation of any of the calculi of Tab. 1, and let $\mathtt{x} \in \{\mathtt{m}, \mathtt{e}\}$. Then, $t \equiv u$ and $t \multimap_{\mathtt{x}} t'$ implies that there exists u' such that $u \multimap_{\mathtt{x}} u'$ and $t' \equiv u'$.*

Proof. See Sect. B of the appendix (page 14). □

The essential property of strong bisimulations is that they can be postponed. In fact, it is immediate to prove the following, which holds for all four calculi:

Lemma 2.3 ($\equiv$ Postponement)**.** *If $t\ (\multimap_{\mathtt{m}} \cup \multimap_{\mathtt{e}} \cup \equiv)^*\ u$ then $t\ (\multimap_{\mathtt{m}} \cup \multimap_{\mathtt{e}})^* \equiv u$ and the number of $\multimap_{\mathtt{m}}$ and $\multimap_{\mathtt{e}}$ steps in the two reduction sequences is exactly the same.*

In the simulation theorems for machines with a global environment (see Sect. 7.1 and Sect. 8) we will also use the following commutation property between substitutions and evaluation contexts via the structural equivalence of every evaluation scheme, proved by an easy induction on the actual definition of evaluation contexts.

Lemma 2.4 (ES Commute with Evaluation Contexts via $\equiv$)**.** *For every evaluation scheme let C denote an evaluation context s.t. $x \notin \mathtt{fv}(C)$ and $\equiv$ be its structural equivalence. Then $C\langle t\rangle[x{\leftarrow}u] \equiv C\langle t[x{\leftarrow}u]\rangle$.*

3. Preliminaries on Abstract Machines.

Codes. All the abstract machines we will consider execute pure λ-terms. In our syntax, these are nothing but terms *without explicit substitutions*. Moreover, while for calculi we work implicitly modulo α, for machines we will *not* consider terms up to α, as the handling of α-equivalence characterizes different approaches to abstract machines. To stress these facts, we use the metavariables $\overline{t}, \overline{u}, \overline{w}, \overline{r}$ for pure λ-terms (not up to α) and $\overline{v}$ for pure values.

States. A machine state s will have various components, of which the first will always be *the code*, *i.e.* a pure λ-term $\overline{t}$. The others (*environment*, *stack*, *dump*) are all considered as lists, whose constructors are the empty list ϵ and the concatenation operator $::$. A state s of a machine is *initial* if its code $\overline{t}$ is closed (*i.e.*, $\mathtt{fv}(\overline{t}) = \emptyset$) and all other components are empty. An *execution* ρ is a sequence of transitions of the machine $s_0 \to^* s$ from an initial state s_0. In that case, we say that s is a *reachable state*, and if $\overline{t}$ is the code of s_0 then $\overline{t}$ is the *initial code* of s.

Invariants. For every machine our study will rely on a lemma about some *dynamic invariants*, *i.e.* some properties of the reachable states that are stable by executions. The lemma is always proved by a straightforward induction on the length of the execution and *the proof is omitted*.

Environments and Closures. There will be two types of machines, those with many *local environments* and those with just one *global environment*. Machines with local environments are based on the mutually recursive definition of *closure* (ranged over by c) and *environment* (e):

$$c \quad ::= \quad (\overline{t}, e) \qquad e \quad ::= \quad \epsilon \mid [x{\leftarrow}c] :: e$$

Global environments are defined by $E ::= \epsilon \mid [x{\leftarrow}\overline{t}] :: E$, and global environment machines will have just one global closure $(\overline{t}, E)$.

Well-Named and Closed Closures. The explicit treatment of α-equivalence, is based on particular representants of α-classes defined via the notion of support. The *support* Δ of codes, environments, and closures is defined by:

- $\Delta(\overline{t})$ is the *multi*set of its bound names (*e.g.* $\Delta(\lambda x.\lambda y.\lambda x.(zx))$ $= [x, x, y]$).
- $\Delta(e)$ is the *multi*set of names captured by e (for example $\Delta([x{\leftarrow}c_1][y{\leftarrow}c_2][x{\leftarrow}c_3]) = [x, x, y]$), and similarly for $\Delta(E)$.
- $\Delta(\overline{t}, e) := \Delta(\overline{t}) + \Delta(e)$ and $\Delta(\overline{t}, E) := \Delta(\overline{t}) + \Delta(E)$.

A code/environment/closure is *well-named* if its support is a set (*i.e.* a multiset with no repetitions). Moreover, a closure $(\overline{t}, e)$ (resp. (t, E)) is *closed* if $\mathtt{fv}(\overline{t}) \subseteq \Delta(e)$ (resp. $\mathtt{fv}(\overline{t}) \subseteq \Delta(E)$).

4. Distilleries

This section presents an abstract, high-level view of the relationship between abstract machines and linear substitution calculi, via the notion of *distillery*.

Definition 4.1. *A* distillery $\mathtt{D} = (\mathtt{M}, \mathtt{C}, \equiv, \underline{\cdot})$ *is given by:*

1. *An* abstract machine $\mathtt{M}$*, given by*
   (a) *a deterministic labeled transition system $\to$ on states s;*
   (b) *a distinguished class of states called* initials *(in bijection with closed λ-terms, and from which applying $\to$ one obtains the* reachable *states);*
   (c) *a partition of its labels as:*
      - *several* commutative *transitions, collectively noted $\to_c$;*
      - *two* principal *transitions, denoted by $\to_m$ and $\to_e$ (for* multiplicative *and* exponential*);*

2. *a* linear substitution calculus $\mathtt{C}$ *given by a pair* $(\multimap_{\mathtt{m}}, \multimap_{\mathtt{e}})$ *of rewriting relations on terms with ES;*
3. *a* structural equivalence $\equiv$ *on terms s.t. it is a strong bisimulation with respect to* $\multimap_{\mathtt{m}}$ *and* $\multimap_{\mathtt{e}}$*;*
4. *a* distillation $\underline{\cdot}$*, i.e. a decoding function from states to terms, s.t. on reachable states:*
   - Commutative*:* $s \to_c s'$ *implies* $\underline{s} \equiv \underline{s'}$.
   - Multiplicative*:* $s \to_m s'$ *implies* $\underline{s} \multimap_{\mathtt{m}}\equiv \underline{s'}$*;*
   - Exponential*:* $s \to_e s'$ *implies* $\underline{s} \multimap_{\mathtt{e}}\equiv \underline{s'}$*;*

Given a distillery, the simulation theorem holds abstractly. Let $|\rho|$ (resp. $|d|$), $|\rho|_m$ (resp. $|d|_m$), $|\rho|_e$ (resp. $|d|_e$), and $|\rho|_p$ denote the number of unspecified, multiplicative, exponential, and principal steps in an execution (resp. derivation).

Theorem 4.2 (Simulation)**.** *Let* $\mathtt{D}$ *be a distillery. Then for every execution* $\rho : s \to^* s'$ *there is a derivation* $d : \underline{s} \multimap^*\equiv \underline{s'}$ *s.t.* $|\rho|_m = |d|_m$*,* $|\rho|_e = |d|_e$*, and* $|\rho|_p = |d|$*.*

Proof. By induction on $|\rho|$ and by the properties of the decoding, it follows that there is a derivation $e : \underline{s}(\multimap\equiv)^*\underline{s'}$ s.t. the number $|\rho|_p = |e|$. The witness d for the statement is obtained by applying the postponement of strong bisimulations (Lemma 2.3) to e. □

Reflection. Given a distillery, one would also expect that reduction in the calculus is reflected in the machine. This result in fact requires two additional abstract properties.

Definition 4.3 (Reflective Distillery)**.** *A distillery is* reflective *when:*

Termination: $\to_c$ terminates *(on reachable states); hence, by determinism, every state* s *has a unique* commutative normal form $\mathtt{nf}_c(s)$*;*

Progress: *if* s *is reachable,* $\mathtt{nf}_c(s) = s$ *and* $\underline{s} \multimap_x t$ *with* $\mathtt{x} \in \{\mathtt{m}, \mathtt{e}\}$*, then there exists* s' *such that* $s \to_{\mathtt{x}} s'$*, i.e.,* s *is not final.*

Then, we may prove the following reflection of steps in full generality:

Proposition 4.4 (Reflection)**.** *Let* $\mathtt{D}$ *be a reflective distillery,* s *be a reachable state, and* $x \in \{m, e\}$*. Then,* $\underline{s} \multimap_{\mathtt{x}} u$ *implies that there exists a state* s' *s.t.* $\mathtt{nf}_c(s) \to_{\mathtt{x}} s'$ *and* $\underline{s'} \equiv u$*.*

In other words, every rewriting step on the calculus can be also performed on the machine, up to commutative transitions.

Proof. The proof is by induction on the number n of transitions leading from s to $\mathtt{nf}_c(s)$.

- *Base case* $n = 0$: by the progress property, we have $s \to_{\mathtt{x}'} s'$ for some state s' and $\mathtt{x}' \in \{\mathtt{m}, \mathtt{e}\}$. By Theorem 4.2, we have $\underline{s} \multimap_{\mathtt{x}'} u' \equiv \underline{s'}$ and we may conclude because $\mathtt{x}' = \mathtt{x}$ and $u' = u$ by determinisim of the calculus (Proposition 2.1).
- *Inductive case* $n > 0$: by hypothesis, we have $s \to_c s_1$. By Theorem 4.2, $\underline{s} \equiv \underline{s_1}$. The hypothesis and the strong bisimulation property (Proposition 2.2) then give us $\underline{s_1} \multimap_{\mathtt{x}} u_1 \equiv u$. But the induction hypothesis holds for s_1, giving us a state s' such that $\mathtt{nf}_c(s_1) \to_{\mathtt{x}} s'$ and $\underline{s'} \equiv u_1 \equiv u$. We may now conclude because $\mathtt{nf}_c(s) = \mathtt{nf}_c(s_1)$. □

The reflection can then be extended to a reverse simulation.

Corollary 4.5 (Reverse Simulation)**.** *Let* $\mathtt{D}$ *be a reflective distillery and* s *an initial state. Given a derivation* $d : \underline{s} \multimap^* t$ *there is an execution* $\rho : s \to^* s'$ *s.t.* $t \equiv \underline{s'}$ *and* $|\rho|_m = |d|_m$*,* $|\rho|_e = |d|_e$*, and* $|\rho|_p = |d|$*.*

Proof. By induction on the length of d, using Proposition 4.4. □

In the following sections we shall introduce abstract machines and distillations for which we will prove that they form reflective distilleries with respect to the calculi of Sect. 2. For each machine we will prove 1) that the decoding is in fact a distillation, and 2) the progress property. *We will instead assume the termination property*, whose proof is delayed to the quantitative study of the second part of the paper, where we will actually prove stronger results, giving explicit bounds.

5. Call-by-Name: the KAM

The Krivine Abstract Machine (KAM) is the simplest machine studied in the paper. A KAM *state* (s) is made out of a closure and of a *stack* (π):

$$\pi \quad ::= \quad \epsilon \mid c :: \pi \qquad\qquad s \quad ::= \quad (c, \pi)$$

For readability, we will use the notation $\overline{t} \mid e \mid \pi$ for a state (c, π) where $c = (\overline{t}, e)$. The transitions of the KAM then are:

$$\begin{array}{c|c|c c c|c|c}
\overline{t}\overline{u} & e & \pi & \to_c & \overline{t} & e & (\overline{u}, e) :: \pi \\
\lambda x.\overline{t} & e & c :: \pi & \to_m & \overline{t} & [x{\leftarrow}c] :: e & \pi \\
x & e & \pi & \to_e & \overline{t} & e' & \pi
\end{array}$$

where $\to_v$ takes place only if $e = e'' :: [x{\leftarrow}(\overline{t}, e')] :: e'''$.

A key point of our study is that environments and stacks rather immediately become contexts of the LSC, through the following decoding:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\underline{\epsilon} & := & \langle\cdot\rangle & \underline{[x{\leftarrow}c] :: e} & := & \underline{e}\langle\langle\cdot\rangle[x{\leftarrow}\underline{c}]\rangle \\
\underline{(\overline{t}, e)} & := & \underline{e}\langle\overline{t}\rangle & \underline{c :: \pi} & := & \underline{\pi}\langle\langle\cdot\rangle\underline{c}\rangle \\
\underline{\overline{t} \mid e \mid \pi} & := & \underline{\pi}\langle\underline{e}\langle\overline{t}\rangle\rangle & & &
\end{array}$$

The decoding satisfies the following static properties, shown by easy inductions on the definition.

Lemma 5.1 (Contextual Decoding)**.** $\underline{e}$ *is a substitution context, and both* $\underline{\pi}$ *and* $\underline{\pi}\langle\underline{e}\rangle$ *are evaluation contexts.*

Next, we need the dynamic invariants of the machine.

Lemma 5.2 (KAM Invariants)**.** *Let* $s = \overline{u} \mid e \mid \pi$ *be a KAM reachable state whose initial code* $\overline{t}$ *is well-named. Then:*

1. Closure*: every closure in* s *is closed;*
2. Subterm*: any code in* s *is a literal subterm of* $\overline{t}$*.*
3. Name*: any closure in* s *is well-named.*
4. Environment Size*: the length of any environment in* s *is bound by* $|\overline{t}|$*.*

Abstract Considerations on Concrete Implementations. The name invariant is the abstract property that allows to avoid α-equivalence in KAM executions. In addition, forbidding repetitions in the support of an environment, it allows to bound the length of any environment with the names in $\overline{t}$, *i.e.* with $|\overline{t}|$. This fact is important, as the static bound on the size of environments guarantees that $\to_e$ and $\to_c$ — the transitions looking-up and copying environments — can be implemented (independently of the chosen concrete representation of terms) in at worst linear time in $|t|$, so that an execution ρ can be implemented in $O(|\rho| \cdot |t|)$. The same will hold for every machine with local environments.

The previous considerations are based on the name and environment size invariants. The closure invariant is used in the progress part of the next theorem, and the subterm invariant is used in the quantitative analysis in Sect. 10 (Theorem 10.3), subsuming the termination condition of reflective distilleries.

Theorem 5.3 (KAM Distillation)**.** $(\mathit{KAM}, \mathtt{Name}, \equiv, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative*: if $s \to_c s'$ then $\underline{s} \equiv \underline{s'}$.*
2. Multiplicative*: if $s \to_m s'$ then $\underline{s} \multimap_{\mathtt{m}} \underline{s'}$;*
3. Exponential*: if $s \to_e s'$ then $\underline{s} \multimap_{\mathtt{e}}\equiv \underline{s'}$;*

Proof. Properties of the decoding:

1. *Commutative*. We have $\overline{t}\overline{u} \mid e \mid \pi \;\to_c\; \overline{t} \mid e \mid (\overline{u}, e) :: \pi$, and:

$$\begin{array}{rcl} \underline{\overline{t}\overline{u} \mid e \mid \pi} & = & \underline{\pi}\langle \underline{e}\langle \overline{t}\overline{u}\rangle\rangle \\ & \equiv_{@}^{*} & \underline{\pi}\langle \underline{e}\langle \overline{t}\rangle \underline{e}\langle \overline{u}\rangle\rangle \;=\; \underline{\overline{t} \mid e \mid (\overline{u}, e) :: \pi} \end{array}$$

2. *Multiplicative*. $\lambda x.\overline{t} \mid e \mid c :: \pi \;\to_m\; \overline{t} \mid [x{\leftarrow}c] :: e \mid \pi$, and

$$\begin{array}{rcl} \underline{\lambda x.\overline{t} \mid e \mid c :: \pi} & = & \underline{\pi}\langle \underline{e}\langle \lambda x.\overline{t}\rangle \underline{c}\rangle \\ & \multimap_{\mathtt{m}} & \underline{\pi}\langle \underline{e}\langle \overline{t}[x{\leftarrow}\underline{c}]\rangle\rangle \\ & = & \underline{\overline{t} \mid [x{\leftarrow}c] :: e \mid \pi} \end{array}$$

   The rewriting step can be applied because by contextual decoding (Lemma 5.1) it takes place in an evaluation context.

3. *Exponential*. $x \mid e' :: [x{\leftarrow}(\overline{t}, e)] :: e'' \mid \pi \;\to_e\; \overline{t} \mid e \mid \pi$, and

$$\begin{array}{rcl} \underline{x \mid e' :: [x{\leftarrow}(\overline{t}, e)] :: e'' \mid \pi} & = & \underline{\pi}\langle \underline{e''}\langle \underline{e'}\langle x\rangle[x{\leftarrow}\underline{e}\langle \overline{t}\rangle]\rangle\rangle \\ & \multimap_{\mathtt{e}} & \underline{\pi}\langle \underline{e''}\langle \underline{e'}\langle \underline{e}\langle \overline{t}\rangle\rangle[x{\leftarrow}\underline{e}\langle \overline{t}\rangle]\rangle\rangle \\ & \equiv_{gc}^{*} & \underline{\pi}\langle \underline{e}\langle \overline{t}\rangle\rangle \\ & = & \underline{\overline{t} \mid e \mid \pi} \end{array}$$

   Note that $e''\langle e'\langle e\langle \overline{t}\rangle\rangle[x{\leftarrow}e\langle \overline{t}\rangle]\rangle \equiv_{gc}^{*} e\langle \overline{t}\rangle$ holds because $e\langle \overline{t}\rangle$ is closed by point 1 of Lemma 5.2, and so all the substitutions around it can be garbage collected.

Termination. Given by (forthcoming) Theorem 10.3 (future proofs of distillery theorems will omit termination).
Progress. Let $s = \overline{t} \mid e \mid \pi$ be a commutative normal form s.t. $\underline{s} \multimap u$. If $\overline{t}$ is

- *an application* $\overline{u}\overline{w}$. Then a $\to_c$ transition applies and s is not a commutative normal form, absurd.
- *an abstraction* $\lambda x.\overline{u}$: if $\pi = \epsilon$ then $\underline{s} = \underline{e}\langle \lambda x.\overline{u}\rangle$, which is $\multimap$-normal, absurd. Hence, a $\to_m$ transition applies.
- *a variable* x: by point 1 of Lemma 5.2.1, we must have $e = e' :: [x{\leftarrow}c] :: e''$, so a $\to_e$ transition applies; □

6. Call-by-Value: the CEK and the LAM

Here we deal with two variants in call-by-value of the KAM, namely Felleisen and Friedman's CEK machine [23] (without control operators) and a machine abstracting Leroy's ZINC machine [32], deemed *Leroy abstract machine* (LAM). They differ on how they behave with respect to applications: the CEK implements left-to-right call-by-value, *i.e.* it first evaluates the function part, the LAM gives instead precedence to arguments, realizing right-to-left call-by-value.

The states of the two machines have the same shape of those of the KAM, *i.e.* they are given by a closure plus a stack. The difference is that they use *call-by-value stacks*, whose elements are labelled either as *functions* or *arguments*, so that the machine may know whether it is launching the evaluation of an argument or it is at the end of such an evaluation. They are re-defined and decoded by (c is a closure):

$$\pi \;::=\; \epsilon \mid \mathbf{f}(c) :: \pi \mid \mathbf{a}(c) :: \pi \qquad\qquad \begin{array}{rcl} \underline{\epsilon} & := & \langle\cdot\rangle \\ \underline{\mathbf{f}(c) :: \pi} & := & \underline{\pi}\langle \underline{c}\langle\cdot\rangle\rangle \\ \underline{\mathbf{a}(c) :: \pi} & := & \underline{\pi}\langle\langle\cdot\rangle \underline{c}\rangle \end{array}$$

The states of both machines are decoded exactly as for the KAM, *i.e.* $\underline{\overline{t} \mid e \mid \pi} := \underline{\pi}\langle \underline{e}\langle \overline{t}\rangle\rangle$.

6.1 Left-to Right Call-by-Value: the CEK machine.

The transitions of the CEK are:

$$\begin{array}{c|c|c|cc|c|c} \overline{t}\overline{u} & e & \pi & \to_{c_1} & \overline{t} & e & \mathbf{a}(\overline{u}, e) :: \pi \\ \overline{v} & e & \mathbf{a}(\overline{u}, e') :: \pi & \to_{c_2} & \overline{u} & e' & \mathbf{f}(\overline{v}, e) :: \pi \\ \overline{v} & e & \mathbf{f}(\lambda x.\overline{t}, e') :: \pi & \to_m & \overline{t} & [x{\leftarrow}(\overline{v}, e)] :: e' & \pi \\ x & e & \pi & \to_e & \overline{t} & e' & \pi \end{array}$$

where $\to_v$ takes place only if $e = e'' :: [x{\leftarrow}(\overline{t}, e')] :: e'''$.

While one can still statically prove that environments decode to substitution contexts, to prove that $\underline{\pi}$ and $\underline{\pi}\langle \underline{e}\rangle$ are evaluation contexts we need the dynamic invariants of the machine.

Lemma 6.1 (CEK Invariants)**.** *Let $s = \overline{u} \mid e \mid \pi$ be a CEK reachable state whose initial code $\overline{t}$ is well-named. Then:*

1. Closure*: every closure in s is closed;*
2. Subterm*: any code in s is a literal subterm of $\overline{t}$;*
3. Value*: any code in e is a value and, for every element of π of the form $\mathbf{f}(\overline{u}, e')$, $\overline{u}$ is a value;*
4. Contextual Decoding*: $\underline{\pi}$ and $\underline{\pi}\langle \underline{e}\rangle$ are left-to-right call-by-value evaluation contexts.*
5. Name*: any closure in s is well-named.*
6. Environment Size*: the length of any environment in s is bound by $|\overline{t}|$.*

We have everything we need:

Theorem 6.2 (CEK Distillation)**.** $(\mathit{CEK}, \mathtt{Value}^{\mathtt{LR}}, \equiv, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative 1*: if $s \to_{c_1} s'$ then $\underline{s} \equiv \underline{s'}$;*
2. Commutative 2*: if $s \to_{c_2} s'$ then $\underline{s} = \underline{s'}$.*
3. Multiplicative*: if $s \to_m s'$ then $\underline{s} \multimap_{\mathtt{m}} \underline{s'}$;*
4. Exponential*: if $s \to_e s'$ then $\underline{s} \multimap_{\mathtt{e}}\equiv \underline{s'}$;*

Proof. Properties of the decoding: in the following cases, evaluation will always takes place under a context that by Lemma 6.1.4 will be a left-to-right call-by-value evaluation context, and similarly structural equivalence will alway be used in a weak context, as it should be.

1. *Commutative 1*. We have $\overline{t}\overline{u} \mid e \mid \pi \;\to_{c_1}\; \overline{t} \mid e \mid \mathbf{a}(\overline{u}, e) :: \pi$, and:

$$\begin{array}{rcl} \underline{\overline{t}\overline{u} \mid e \mid \pi} & = & \underline{\pi}\langle \underline{e}\langle \overline{t}\overline{u}\rangle\rangle \quad \equiv_{@}^{*} \\ & & \underline{\pi}\langle \underline{e}\langle \overline{t}\rangle \underline{e}\langle \overline{u}\rangle\rangle \;=\; \underline{\overline{t} \mid e \mid \mathbf{a}(\overline{u}, e) :: \pi} \end{array}$$

2. *Commutative 2*. We have $\overline{v} \mid e \mid \mathbf{a}(\overline{u}, e') :: \pi \;\to_{c_2}\; \overline{u} \mid e' \mid \mathbf{f}(\overline{v}, e) :: \pi$, and:

$$\begin{array}{rcl} \underline{\overline{v} \mid e \mid \mathbf{a}(\overline{u}, e') :: \pi} & = & \underline{\pi}\langle \underline{e}\langle \overline{v}\rangle \underline{e'}\langle \overline{u}\rangle\rangle \quad = \\ & & \underline{\overline{u} \mid e' \mid \mathbf{f}(\overline{v}, e) :: \pi} \end{array}$$

3. *Multiplicative*. We have $\overline{v} \mid e \mid \mathbf{f}(\lambda x.\overline{t}, e') :: \pi \;\to_m\; \overline{u} \mid [x{\leftarrow}(\overline{v}, e)] :: e' \mid \pi$, and:

$$\begin{array}{rcl} \underline{\overline{v} \mid e \mid \mathbf{f}(\lambda x.\overline{t}, e') :: \pi} & = & \underline{\pi}\langle \underline{e'}\langle \lambda x.\overline{t}\rangle \underline{e}\langle \overline{v}\rangle\rangle \quad \multimap_{\mathtt{m}} \\ & & \underline{\pi}\langle \underline{e'}\langle \overline{t}[x{\leftarrow}\underline{e}\langle \overline{v}\rangle]\rangle\rangle \quad = \\ & & \underline{\overline{t} \mid [x{\leftarrow}(\overline{v}, e)] :: e' \mid \pi} \end{array}$$

4. *Exponential*. Let $e = e'' :: [x{\leftarrow}(\overline{t}, e')] :: e'''$. We have $x \mid e \mid \pi \;\to_e\; \overline{t} \mid e' \mid \pi$, and:

$$\begin{array}{lcll}
\underline{x \mid e \mid \pi} & = & \pi\langle e\langle x\rangle\rangle & = \\
& & \underline{\pi}\langle \underline{e'''}\langle \underline{e''}\langle x\rangle[x\leftarrow \underline{e'}\langle \bar{t}\rangle]\rangle\rangle & \multimap_{\mathtt{e}} \\
& & \underline{\pi}\langle \underline{e'''}\langle \underline{e'}\langle \underline{e''}\langle \bar{t}\rangle[x\leftarrow \bar{t}]\rangle\rangle\rangle & \equiv^*_{gc} \\
& & \underline{\pi}\langle \underline{e'}\langle \bar{t}\rangle\rangle & = \quad \underline{\bar{t} \mid e' \mid \pi}
\end{array}$$

We can apply $\multimap_{\mathtt{e}}$ since by Lemma 6.1.3, $\bar{t}$ is a value. We also use that by Lemma 6.1.1, $\underline{e'}\langle \bar{t}\rangle$ is a closed term to ensure that $\underline{e''}$ and $\underline{e'''}$ can be garbage collected.

Progress. Let $s = \bar{t} \mid e \mid \pi$ be a commutative normal form s.t. $\underline{s} \multimap u$. If $\bar{t}$ is

- *an application* $\overline{uw}$. Then a $\rightarrow_{c_1}$ transition applies and s is not a commutative normal form, absurd.
- *an abstraction* $\bar{v}$: by hypothesis, π cannot be of the form $\mathbf{a}(c) :: \pi'$. Suppose it is equal to ϵ. We would then have $\underline{s} = \underline{e}\langle \bar{v}\rangle$, which is a call-by-value normal form, because $\underline{e}$ is a substitution context. This would contradict our hypothesis, so π must be of the form $\mathbf{f}(\bar{u}, e') :: \pi'$. By point 3 of Lemma 6.1, $\bar{u}$ is an abstraction, hence a $\rightarrow_{\lambda_1}$ transition applies.
- *a variable* x: by point 1 of Lemma 6.1, e must be of the form $e' :: [x\leftarrow c] :: e''$, so a $\rightarrow_v$ transition applies; □

6.2 Right-to-Left Call-by-Value: the Leroy Abstract Machine

The transitions of the LAM are:

$$\begin{array}{c|c|c c c|c|c}
\bar{t}\bar{u} & e & \pi & \rightarrow_{c_1} & \bar{u} & e & \mathbf{f}(\bar{t}, e) :: \pi \\
\bar{v} & e & \mathbf{f}(\bar{t}, e') :: \pi & \rightarrow_{c_2} & \bar{t} & e' & \mathbf{a}(\bar{v}, e) :: \pi \\
\lambda x.\bar{t} & e & \mathbf{a}(c) :: \pi & \rightarrow_{m} & \bar{t} & [x\leftarrow c] :: e & \pi \\
x & e & \pi & \rightarrow_{e} & \bar{t} & e' & \pi
\end{array}$$

where $\rightarrow_v$ takes place only if $e = e'' :: [x\leftarrow(\bar{t}, e')] :: e'''$.

We omit all the proofs (that can be found in the appendix, page 22) because they are minimal variations on those for the CEK.

Lemma 6.3 (LAM Invariants). *Let $s = \bar{u} \mid e \mid \pi$ be a LAM reachable state whose initial code $\bar{t}$ is well-named. Then:*

1. Closure*: every closure in s is closed;*
2. Subterm*: any code in s is a literal subterm of $\bar{t}$;*
3. Value*: any code in e is a value and, for every element of π of the form $\mathbf{a}(\bar{u}, e')$, $\bar{u}$ is a value;*
4. Contexts Decoding*: $\underline{\pi}$ and $\underline{\pi}\langle\underline{e}\rangle$ are right-to-left call-by-value evaluation contexts.*
5. Name*: any closure in s is well-named.*
6. Environment Size*: the length of any environment in s is bound by $|\bar{t}|$.*

Theorem 6.4 (LAM Distillation). *(LAM, $\mathtt{Value}^{\mathtt{RL}}$, $\equiv$, $\underline{\cdot}$) is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative 1*: if $s \rightarrow_{c_1} s'$ then $\underline{s} \equiv \underline{s'}$;*
2. Commutative 2*: if $s \rightarrow_{c_2} s'$ then $\underline{s} = \underline{s'}$.*
3. Multiplicative*: if $s \rightarrow_m s'$ then $\underline{s} \multimap_{\mathtt{m}} \underline{s'}$;*
4. Exponential*: if $s \rightarrow_e s'$ then $\underline{s} \multimap_{\mathtt{e}} \equiv \underline{s'}$;*

7. Towards Call-by-Need: the MAM and the Split CEK

In this section we study two further machines:

1. *The Milner Abstract Machine (MAM)*, that is a variation over the KAM with only one global environment and without the concept of closure. Essentially, it unveils the content of distance rules at the machine level.
2. *The Split CEK (SCEK)*, obtained disentangling the two uses of the stack (for arguments and for functions) in the CEK. The split CEK can be seen as a simplification of Landin's SECD machine [30].

The ideas at work in these two case studies will be combined in the next section, obtaining a new simple call-by-need machine.

7.1 Milner Abstract Machine

The linear substitution calculus suggests the design of a simpler version of the KAM, the Milner Abstract Machine (MAM), that avoids the concept of closure. At the language level, the idea is that, by repeatedly applying the axioms $\equiv_{dup}$ and $\equiv_{@}$ of the structural equivalence, explicit substitutions can be folded and brought *outside*. At the machine level, the local environments in the closures are replaced by just one global environment that closes the code and the stack, as well as the global environment itself.

Of course, naively turning to a global environment breaks the well-named invariant of the machine. This point is addressed using an α-renaming in the variable transition, *i.e.* when substitution takes place. Here we employ the global environments E of Sect. 3 and we redefine stacks as $\pi ::= \epsilon \mid \bar{t} :: \pi$. A state of the MAM is given by a code $\bar{t}$, a stack π and a global environment E. Note that the code and the stack together now form a code.

The transitions of the MAM are:

$$\begin{array}{c|c|c c c|c|c}
\bar{t}\bar{u} & \pi & E & \rightarrow_{c} & \bar{t} & \bar{u} :: \pi & E \\
\lambda x.\bar{t} & \bar{u} :: \pi & E & \rightarrow_{m} & \bar{t} & \pi & [x\leftarrow\bar{u}] :: E \\
x & \pi & E & \rightarrow_{e} & \bar{t}^{\alpha} & \pi & E
\end{array}$$

where $\rightarrow_v$ takes place only if $E = E''\langle E'[x\leftarrow\bar{t}]\rangle$ and $\bar{t}^{\alpha}$ is a well-named code α-equivalent to $\bar{t}$ and s.t. any bound name in $\bar{t}^{\alpha}$ is fresh with respect to those in π and E[1].

The decoding of a MAM state $\bar{t} \mid \pi \mid E$ is similar to the decoding of a KAM state, but the stack and the environment context are applied in reverse order (this is why stack and environment in MAM states are swapped with respect to KAM states):

$$\begin{array}{rcl@{\qquad}rcl}
\underline{\epsilon} & := & \langle\cdot\rangle & \underline{[x\leftarrow\bar{t}] :: E} & := & \underline{E}\langle\langle\cdot\rangle[x\leftarrow\bar{t}]\rangle \\
\underline{\bar{t} :: \pi} & := & \underline{\pi}\langle\langle\cdot\rangle\bar{t}\rangle & \underline{\bar{t} \mid \pi \mid E} & := & \underline{E}\langle\underline{\pi}\langle\bar{t}\rangle\rangle
\end{array}$$

We call *global closure* associated to state $\bar{t} \mid \pi \mid E$ the pair $(\bar{t}\underline{\pi}, E)$.

As for the KAM, the decoding of contexts can be done statically, *i.e.* it does not need dynamic invariants.

Lemma 7.1 (Contextual Decoding). *$\underline{E}$ is a substitution context, and both $\underline{\pi}$ and $\underline{\pi}\langle\underline{E}\rangle$ are evaluation contexts.*

For the dynamic invariants we need a different notion of closed closure.

Definition 7.2. *Given a global environment E and a code $\bar{t}$, we define by mutual induction two predicates* E is closed *and* $(\bar{t}, E)$ is closed *as follows:*

$$\begin{array}{rcl}
& & \epsilon \textit{ is closed} \\
(\bar{t}, E) \textit{ is closed} & \Longrightarrow & [x\leftarrow t] :: E \textit{ is closed} \\
\mathtt{fv}(\bar{t}) \subseteq \Delta(E) \wedge E \textit{ is closed} & \Longrightarrow & (\bar{t}, E) \textit{ is closed}
\end{array}$$

The dynamic invariants are:

Lemma 7.3 (MAM invariants). *Let $s = \bar{u} \mid \pi \mid E$ be a MAM state reached by an execution ρ of initial well-named code $\bar{t}$. Then:*

[1] The well-named invariant can be restored also in another way. One can simply substitute $\bar{t}$ (instead of $\bar{t}^{\alpha}$) but modify $\rightarrow_{\lambda}$ as follows (with y fresh):

$$\lambda x.\bar{t} \mid \bar{u} :: \pi \mid E \quad \rightarrow_{\lambda} \quad \bar{t}\{x\leftarrow y\} \mid \pi \mid [y\leftarrow\bar{u}] :: E$$

$\overline{t}\overline{u}$	e	π	D	$\rightarrow_{c_1}$	$\overline{t}$	e	$(\overline{u},e)::\pi$	D
$\overline{v}$	e	$(\overline{t},e')::\pi$	D	$\rightarrow_{c_2}$	$\overline{t}$	e'	ϵ	$((\overline{v},e),\pi)::D$
$\overline{v}$	e	ϵ	$((\lambda x.\overline{t},e'),\pi)::D$	$\rightarrow_m$	$\overline{t}$	$[x\leftarrow(\overline{v},e)]::e'$	π	D
x	$e::[x\leftarrow(\overline{v},e')]::e''$	π	D	$\rightarrow_e$	$\overline{v}$	e'	π	D

Figure 2. The Split CEK, aka the revisited SECD.

1. Global Closure: *the global closure* $(\overline{t}\underline{\pi},E)$ *of* s *is closed;*
2. Subterm: *any code in* s *is a literal subterm of* $\overline{t}$*;*
3. Names: *the global closure of* s *is well-named;*
4. Environment Size: *the length of the global environment in* s *is bound by* $|\rho|_m$.

Abstract Considerations on Concrete Implementations. Note the new environment size invariant, whose bound is laxer than for local environment machines. Let ρ be a execution of initial code $\overline{t}$. If one implements $\rightarrow_e$ looking for x in E sequentially, then each $\rightarrow_e$ transition has cost $|\rho|_m$ (more precisely, linear in the number of preceding $\rightarrow_m$ transitions) and the cost of implementing ρ is easily seen to become quadratic in $|\rho|$. An efficient implementation would then employ a representation of codes such that variables are pointers, so that looking for x in E takes constant time. The name invariant guarantees that variables can indeed taken as pointers, as there is no name clash. Note that the cost of a $\rightarrow_e$ transition is not constant, as the renaming operation actually makes $\rightarrow_e$ linear in $|t|$ (by the subterm invariant). So, assuming a pointer-based representation, ρ can be implemented in time $O(|\rho|\cdot|\overline{t}|)$, as for local machines, and the same will hold for every global environment machine.

Theorem 7.4 (MAM Distillation)**.** $(MAM, \texttt{Name}, \equiv, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state* s *we have:*

1. Commutative*: if* $s \rightarrow_{@} s'$ *then* $\underline{s} = \underline{s'}$*;*
2. Multiplicative*: if* $s \rightarrow_{\lambda} s'$ *then* $\underline{s} \multimap_{\mathtt{m}}\equiv \underline{s'}$*;*
3. Exponential*: if* $s \rightarrow_{v} s'$ *then* $\underline{s} \multimap_{\mathtt{e}}=_{\alpha} \underline{s'}$*.*

Proof. Properties of the decoding (*progress* is as for the KAM):

1. *Commutative*. In contrast to the KAM, $\rightarrow_c$ gives a true identity:
$$\underline{\overline{t}\overline{u} \mid \pi \mid E} \quad = \quad \underline{E}\langle\underline{\pi}\langle\overline{t}\overline{u}\rangle\rangle \quad = \quad \underline{\overline{t} \mid \overline{u}::\pi \mid E}$$
2. *Multiplicative*. Since substitutions and evaluation contexts commute via $\equiv$ (Lemma 2.4), $\rightarrow_m$ maps to:
$$\begin{array}{rcll} \underline{\lambda x.\overline{t} \mid \overline{u}::\pi \mid E} & = & \underline{E}\langle\underline{\pi}\langle(\lambda x.\overline{t})\overline{u}\rangle\rangle & \multimap_{\mathtt{m}} \\ & & \underline{E}\langle\underline{\pi}\langle\overline{t}[x\leftarrow\overline{u}]\rangle\rangle & \equiv_{Lem.2.4} \\ & & \underline{E}\langle\underline{\pi}\langle\overline{t}\rangle[x\leftarrow\overline{u}]\rangle & = \\ & & \underline{\overline{t} \mid \pi \mid [x\leftarrow\overline{u}]::E} & \end{array}$$
3. *Exponential*. The erasure of part of the environment of the KAM is replaced by an explicit use of α-equivalence:
$$\begin{array}{rcll} \underline{x \mid \pi \mid E::[x\leftarrow\overline{u}]::E'} & = & \underline{E'}\langle\underline{E}\langle\underline{\pi}\langle x\rangle\rangle[x\leftarrow\overline{u}]\rangle & \multimap_{\mathtt{e}} \\ & & \underline{E'}\langle\underline{E}\langle\underline{\pi}\langle\overline{u}\rangle\rangle[x\leftarrow\overline{u}]\rangle & =_{\alpha} \\ & & \underline{E'}\langle\underline{E}\langle\underline{\pi}\langle\overline{u}^{\alpha}\rangle\rangle[x\leftarrow\overline{u}]\rangle & = \\ & & \underline{\overline{u}^{\alpha} \mid \pi \mid E::[x\leftarrow\overline{u}]::E'} & \end{array}$$

□

Digression about $\equiv$. Note that in the distillation theorem structural equivalence is used only to commute with stacks. The calculus and the machine in fact form a distillery also with respect to the following simpler notion of structural equivalence. Let $\equiv_{\text{MAM}}$ be the smallest equivalence relation generated by the closure by (call-by-name) evaluation contexts of the axiom $\equiv_{@l}$ in Fig. 1 (page 4). The next lemma guarantees that $\equiv_{\text{MAM}}$ is a strong bisimulation (the proof is in the appendix, page 23), and so $\equiv_{\text{MAM}}$ provides another MAM distillery.

Lemma 7.5. $\equiv_{MAM}$ *is a strong bisimulation with respect to* $\multimap$*.*

7.2 The Split CEK, or Revisiting the SECD Machine

For the CEK machine we proved that the stack, that collects both arguments and functions, decodes to an evaluation context (Lemma 6.1.4). The new CBV machine in Fig. 2, deemed *Split CEK*, has two stacks: one for arguments and one for functions. Both will decode to evaluation contexts. The argument stack is identical to the stack of the KAM, and, accordingly, will decode to an applicative context. Roughly, the function stack decodes to contexts of the form $H\langle v\langle\cdot\rangle\rangle$. More precisely, an entry of the function stack is a pair (c,π), where c is a closure $(\overline{v},e)$, and the three components $\overline{v}$, e, and π together correspond to the evaluation context $\underline{\pi}\langle\underline{e}\langle\overline{v}\langle\cdot\rangle\rangle\rangle$. For the acquainted reader, this new stack corresponds to the *dump* of Landin's SECD machine [30].

Let us explain the main idea. Whenever the code is an abstraction $\overline{v}$ and the argument stack π is non-empty (*i.e.* $\pi = c::\pi'$), the machine saves the active closure, given by current code $\overline{v}$ and environment e, and the tail of the stack π' by pushing a new entry $((\overline{v},e),\pi')$ on the dump, and then starts evaluating the first closure c of the stack. The syntax for dumps then is

$$D ::= \epsilon \mid (c,\pi)::D$$

Every dump decodes to a context according to:

$$\underline{\epsilon} \quad := \quad \langle\cdot\rangle \qquad\qquad \underline{((\overline{v},e),\pi)::D} \quad := \quad \underline{D}\langle\underline{\pi}\langle\underline{e}\langle\overline{v}\langle\cdot\rangle\rangle\rangle\rangle$$

The decoding of terms, environments, closures, and stacks is as for the KAM. The decoding of states is defined as $\underline{\overline{t} \mid e \mid \pi \mid D} := \underline{D}\langle\underline{\pi}\langle\underline{e}\langle\overline{t}\rangle\rangle\rangle$. The proofs for the Split CEK are in the appendix (page 23).

Lemma 7.6 (Split CEK Invariants)**.** *Let* $s = \overline{u} \mid e \mid \pi \mid D$ *be a Split CEK reachable state whose initial code* $\overline{t}$ *is well-named. Then:*

1. Closure: *every closure in* s *is closed;*
2. Subterm: *any code in* s *is a literal subterm of* $\overline{t}$*;*
3. Value: *the code of any closure in the dump or in any environment in* s *is a value;*
4. Contextual Decoding*:* $\underline{D}$*,* $\underline{D}\langle\underline{\pi}\rangle$*, and* $\underline{D}\langle\underline{\pi}\langle\underline{e}\rangle\rangle$ *are left-to-right call-by-value evaluation context.*
5. Name*: any closure in* s *is well-named.*
6. Environment Size*: the length of any environment in* s *is bound by* $|\overline{t}|$*.*

Theorem 7.7 (Split CEK Distillation)**.** (Split CEK, $\texttt{Value}^{\texttt{LR}}$, $\underline{\cdot}$) *is a reflective distillery. In particular, on a reachable state* s *we have:*

1. Commutative 1*: if* $s \rightarrow_{c_1} s'$ *then* $\underline{s} \equiv \underline{s'}$*;*
2. Commutative 2*: if* $s \rightarrow_{c_2} s'$ *then* $\underline{s} \equiv \underline{s'}$*;*
3. Multiplicative*: if* $s \rightarrow_{m} s'$ *then* $\underline{s} \multimap_{\mathtt{m}} \underline{s'}$*;*
4. Exponential*: if* $s \rightarrow_{e} s'$ *then* $\underline{s} \multimap_{\mathtt{e}}\equiv \underline{s'}$*.*

$$
\begin{array}{c|c|c|c c c|c|c|c}
\overline{t}\overline{u} & \pi & D & E & \rightarrow_{c_1} & \overline{t} & \overline{u} :: \pi & D & E \\
\lambda x.\overline{t} & \overline{u} :: \pi & D & E & \rightarrow_{m} & \overline{t} & \pi & D & [x{\leftarrow}\overline{u}] :: E \\
x & \pi & D & E_1 :: [x{\leftarrow}\overline{t}] :: E_2 & \rightarrow_{c_2} & \overline{t} & \epsilon & (E_1, x, \pi) :: D & E_2 \\
\overline{v} & \epsilon & (E_1, x, \pi) :: D & E_2 & \rightarrow_{e} & \overline{v}^{\alpha} & \pi & D & E_1 :: [x{\leftarrow}\overline{v}] :: E_2
\end{array}
$$

Figure 3. The Wadsworth Abstract Machine (WAM).

8. Call-by-Need: the WAM and the Merged WAM

A new abstract machine for call-by-need, deemed *Wadsworth Abstract Machine* (WAM), is shown in Fig. 3. It is obtained from the KAM by two tweaks:

1. It uses the dump-like approach of the Split CEK/SECD to evaluate inside explicit substitutions
2. It uses the global environment approach of the MAM to implement memoization;

Whenever the code is a variable x and the environment has the form $E_1 :: [x{\leftarrow}\overline{t}] :: E_2$, the machine jumps to evaluate $\overline{t}$ saving the prefix of the environment E_1, the variable x on which it will substitute the result of evaluating $\overline{t}$, and the stack π. In Sect. 9, we will present a variant of the WAM that avoids the splitting of the environment saving E_1 in a dump entry.

The syntax for dumps is

$$D ::= \epsilon \mid (E, x, \pi) :: D$$

Every dump stack decodes to a context according to:

$$\underline{\epsilon} \;:=\; \langle\cdot\rangle \qquad \underline{(E, x, \pi) :: D} \;:=\; \underline{E}\langle\underline{D}\langle\underline{\pi}\langle x\rangle\rangle\rangle[x{\leftarrow}\langle\cdot\rangle]$$

The decoding of terms, environments, and stacks is defined as for the KAM. The decoding of states is defined as $\underline{\overline{t} \mid \pi \mid D \mid E} := \underline{E}\langle\underline{D}\langle\underline{\pi}\langle\overline{t}\rangle\rangle\rangle$. The decoding of contexts is static:

Lemma 8.1 (Contextual Decoding)**.** *$\underline{D}$, $\underline{D}\langle\underline{\pi}\rangle$, $\underline{E}\langle\underline{D}\rangle$, and $\underline{E}\langle\underline{D}\langle\underline{\pi}\rangle\rangle$ are call-by-need evaluation contexts.*

Closed closures are defined as for the MAM. Given a state $s = \overline{t} \mid \pi \mid D \mid E_0$ with $D = (E_1, x_1, \pi_1) :: \ldots :: (E_n, x_n, \pi_n)$, its closures are $(\underline{\pi}\langle\overline{t}\rangle, E_0)$ and, for $i \in \{1, \ldots, n\}$,

$$(\underline{\pi_i}\langle x_i\rangle, E_i :: [x_i{\leftarrow}\underline{\pi_{i-1}}\langle x_{i-1}\rangle] :: \ldots :: [x_1{\leftarrow}\underline{\pi}\langle\overline{t}\rangle] :: E_0).$$

The dynamic invariants are:

Lemma 8.2 (WAM invariants)**.** *Let $s = \overline{t} \mid \pi \mid D \mid E_0$ be a WAM reachable state whose initial code $\overline{t}$ is well-named, and s.t. $D = (E_1, x_1, \pi_1) :: \ldots :: (E_n, x_n, \pi_n)$. Then:*

1. Global Closure: *the closures of s are closed;*
2. Subterm: *any code in s is a literal subterm of $\overline{t}$;*
3. Names: *the closures of s are well-named.*

For the properties of the decoding function please note that, as defined in Sect. 2, the structural congruence for call-by-need is different from before.

Theorem 8.3 (WAM Distillation)**.** $(\mathrm{WAM}, \mathtt{Need}, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative 1*: if $s \rightarrow_{c_1} s'$ then $\underline{s} = \underline{s'}$;*
2. Commutative 2*: if $s \rightarrow_{c_2} s'$ then $\underline{s} = \underline{s'}$;*
3. Multiplicative*: if $s \rightarrow_{m} s'$ then $\underline{s} \multimap_{\mathtt{m}} \equiv \underline{s'}$;*
4. Exponential*: if $s \rightarrow_{e} s'$ then $\underline{s} \multimap_{\mathtt{e}} =_{\alpha} \underline{s'}$.*

Proof. 1. *Commutative 1.*

$$\underline{\overline{t}\,\overline{u} \mid \pi \mid D \mid E} = \underline{E}\langle\underline{D}\langle\underline{\pi}\langle\overline{t}\,\overline{u}\rangle\rangle\rangle = \underline{\overline{t} \mid \overline{u} :: \pi \mid D \mid E}$$

2. *Commutative 2*:

$$
\begin{array}{lcl}
\underline{x \mid \pi \mid D \mid E_1 :: [x{\leftarrow}\overline{t}] :: E_2} & = & \underline{E_2}\langle\underline{E_1}\langle\underline{D}\langle\underline{\pi}\langle x\rangle\rangle\rangle[x{\leftarrow}\overline{t}]\rangle \quad = \\
& & \underline{\overline{t} \mid \epsilon \mid (E_1, x, \pi) :: D \mid E_2}
\end{array}
$$

3. *Multiplicative*.

$$
\begin{array}{lcll}
\underline{\lambda x.\overline{t} \mid \overline{u} :: \pi \mid D \mid E} & = & \underline{E}\langle\underline{D}\langle\underline{\pi}\langle(\lambda x.\overline{t})\,\overline{u}\rangle\rangle\rangle & \multimap_{\mathtt{m}} \\
& & \underline{E}\langle\underline{D}\langle\underline{\pi}\langle\overline{t}[x{\leftarrow}\overline{u}]\rangle\rangle\rangle & \equiv_{Lem.2.4} \\
& & \underline{E}\langle\underline{D}\langle\underline{\pi}\langle\overline{t}\rangle\rangle[x{\leftarrow}\overline{u}]\rangle & = \\
& & \underline{\overline{t} \mid \pi \mid D \mid [x{\leftarrow}\overline{u}] :: E} &
\end{array}
$$

Note that to apply Lemma 2.4 we use the global closure invariant, as $\overline{u}$, being on the stack, is closed by E and so $\underline{D}$ does not capture its free variables.

4. *Exponential*.

$$
\begin{array}{lcll}
\underline{\overline{v} \mid \epsilon \mid (E_1, x, \pi) :: D \mid E_2} & = & \underline{E_2}\langle\underline{E_1}\langle\underline{D}\langle\underline{\pi}\langle x\rangle\rangle\rangle[x{\leftarrow}\overline{v}]\rangle & \multimap_{\mathtt{e}} \\
& & \underline{E_2}\langle\underline{E_1}\langle\underline{D}\langle\underline{\pi}\langle\overline{v}\rangle\rangle\rangle[x{\leftarrow}\overline{v}]\rangle & =_{\alpha} \\
& & \underline{E_2}\langle\underline{E_1}\langle\underline{D}\langle\underline{\pi}\langle\overline{v}^{\alpha}\rangle\rangle\rangle[x{\leftarrow}\overline{v}]\rangle & = \\
& & \underline{\overline{v}^{\alpha} \mid \pi \mid D \mid E_1 :: [x{\leftarrow}\overline{v}] :: E_2} &
\end{array}
$$

Progress. Let $s = \overline{t} \mid \pi \mid D \mid E$ be a commutative normal form s.t. $\underline{s} \multimap u$. If $\overline{t}$ is

1. *an application* $\overline{u}\overline{w}$. Then a $\rightarrow_{c_1}$ transition applies and s is not a commutative normal form, absurd.
2. *an abstraction* v. The decoding $\underline{s}$ is of the form $\underline{E}\langle\underline{D}\langle\underline{\pi}\langle v\rangle\rangle\rangle$. The stack π and the dump D cannot both be empty, since then $\underline{s} = \underline{E}\langle v\rangle$ would be normal. So either the stack is empty and a $\rightarrow_e$ transition applies, or the stack is not empty and a $\rightarrow_m$ transition applies.
3. *a variable* x. By Lemma 8.2.1 it must be bound by E, so a $\rightarrow_{c_2}$ transition applies, and s is not a commutative normal form, absurd. □

8.1 The Merged WAM, or Revisiting the Lazy KAM

Splitting the stack of the CEK machine in two we obtained a simpler form of the SECD machine. In this section we apply to the WAM the reverse transformation. The result is a machine, deemed *merged WAM*, having only one stack and that can be seen as a simpler version of the lazy KAM.

To distinguish the two kinds of objects on the stack we use a marker, as for the CEK and the LAM. Formally, the syntax for stacks is:

$$\pi ::= \epsilon \mid \mathbf{a}(\overline{t}) :: \pi \mid \mathbf{h}(E, x) :: \pi$$

where $\mathbf{a}(\overline{t})$ denotes a term to be used as an argument (as for the CEK) and $\mathbf{h}(E, x, \pi)$ is morally an entry of the dump of the WAM, where however there is no need to save the current stack. The transitions of the Merged WAM are in Fig. 4.

The decoding is defined as follows

$$
\begin{array}{rcl}
\underline{\epsilon} & := & \langle\cdot\rangle \\
\underline{[x{\leftarrow}\overline{t}] :: E} & := & \underline{E}\langle\langle\cdot\rangle[x{\leftarrow}\overline{t}]\rangle \\
\underline{\mathbf{h}(E, x) :: \pi} & := & \underline{E}\langle\underline{\pi}\langle x\rangle\rangle[x{\leftarrow}\langle\cdot\rangle] \\
\underline{\mathbf{a}(\overline{t}) :: \pi} & := & \underline{\pi}\langle\langle\cdot\rangle\overline{t}\rangle \\
\underline{\overline{t} \mid \pi \mid E} & := & \underline{E}\langle\underline{\pi}\langle\overline{t}\rangle\rangle
\end{array}
$$

$$
\begin{array}{c|c|ccc|c|c}
\bar{t}\bar{u} & \pi & E & \rightarrow_{c_1} & \bar{t} & \mathbf{a}(\bar{u}) :: \pi & E \\
\lambda x.\bar{t} & \mathbf{a}(\bar{u}) :: \pi & E & \rightarrow_{m} & \bar{t} & \pi & [x{\leftarrow}\bar{u}] :: E \\
x & \pi & E_1 :: [x{\leftarrow}\bar{t}] :: E_2 & \rightarrow_{c_2} & \bar{t} & \mathbf{h}(E_1, x) :: \pi & E_2 \\
\bar{v} & \mathbf{h}(E_1, x) :: \pi & E_2 & \rightarrow_{e} & \bar{v}^{\alpha} & \pi & E_1 :: [x{\leftarrow}\bar{v}] :: E_2
\end{array}
$$

Figure 4. The Merged WAM.

$$
\begin{array}{c|c|c|ccc|c|c|c}
\bar{t}\bar{u} & \pi & D & E & \rightarrow_{c_1} & \bar{t} & \bar{u} :: \pi & D & E \\
\lambda x.\bar{t} & \bar{u} :: \pi & \epsilon & E & \rightarrow_{m_1} & \bar{t} & \pi & \epsilon & [x{\leftarrow}\bar{u}] :: E \\
\lambda x.\bar{t} & \bar{u} :: \pi & (y, \pi') :: D & E_1 :: [y{\leftarrow}\square] :: E_2 & \rightarrow_{m_2} & \bar{t} & \pi & (y, \pi') :: D & E_1 :: [y{\leftarrow}\square] :: [x{\leftarrow}\bar{u}] :: E_2 \\
x & \pi & D & E_1 :: [x{\leftarrow}\bar{t}] :: E_2 & \rightarrow_{c_2} & \bar{t} & \epsilon & (x, \pi) :: D & E_1 :: [x{\leftarrow}\square] :: E_2 \\
\bar{v} & \epsilon & (x, \pi) :: D & E_1 :: [x{\leftarrow}\square] :: E_2 & \rightarrow_{e} & \bar{v}^{\alpha} & \pi & D & E_1 :: [x{\leftarrow}\bar{v}] :: E_2
\end{array}
$$

Figure 5. The Pointing WAM.

Lemma 8.4 (Contextual Decoding)**.** *$\underline{\pi}$ and $\underline{E}\langle\underline{\pi}\rangle$ are call-by-need evaluation contexts.*

The dynamic invariants of the Merged WAM are exactly the same of the WAM, with respect to an analogous set of closures associated to a state (whose exact definition is omitted). The proof of the following theorem — almost identical to that of the WAM — is in the appendix (page 23).

Theorem 8.5 (Merged WAM Distillation)**.** $(\text{Merged WAM}, \mathtt{Need}, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative 1*: if $s \rightarrow_{c_1} s'$ then $\underline{s} = \underline{s'}$;*
2. Commutative 2*: if $s \rightarrow_{c_2} s'$ then $\underline{s} = \underline{s'}$;*
3. Multiplicative*: if $s \rightarrow_{m} s'$ then $\underline{s} \multimap_{\mathtt{m}}\equiv \underline{s'}$;*
4. Exponential*: if $s \rightarrow_{e} s'$ then $\underline{s} \multimap_{\mathtt{e}}=_{\alpha} \underline{s'}$.*

9. The Pointing WAM, or Revisiting Sestoft's AM

In the WAM, the global environment is divided between the environment of the machine and the entries of the dump. On one hand, this choice makes the decoding very natural. On the other hand, one would like to keep the environment in just one place, letting the dump only collect variables and stacks. This is what we do here, exploiting the fact that variable names can be taken as pointers (see the discussion after the invariants in Sect. 7.1). The new machine, called Pointing WAM, is in Fig. 5, and uses a new dummy constant $\square$ for the substitutions whose variable is in the dump. It can be seen as a simpler version of Sestoft's abstract machine [39]. Dumps and environments (called *hyperstacks* and *heap* by Sestoft) are defined by:

$$
\begin{array}{rcl}
D & ::= & \epsilon \mid (x, \pi) :: D \\
E & ::= & \epsilon \mid [x{\leftarrow}\bar{t}] :: E \mid [x{\leftarrow}\square] :: E
\end{array}
$$

A substitution of the form $[x{\leftarrow}\square]$ is *dumped*, and we also say that x is dumped.

Note that the variables of the entries in D appear in reverse order with respect to the corresponding substitutions in E. We will show that fact is an invariant, called *duality*.

Definition 9.1 (Duality $E \bot D$)**.** *Duality $E \bot D$ between environments and dumps is defined by*

1. *$\epsilon \bot \epsilon$;*
2. *$E :: [x{\leftarrow}\bar{t}] \bot D$ if $E \bot D$;*
3. *$E :: [x{\leftarrow}\square] \bot (x, \pi) :: D$ if $E \bot D$.*

Note that in a dual pair the environment is always long at least as the dump. A dual pair $E \bot D$ decodes to a context as follows:

$$
\begin{array}{rcl}
\underline{(E, \epsilon)} & := & \underline{E} \\
\underline{(E :: [x{\leftarrow}\square], (x, \pi) :: D)} & := & \underline{(E, D)}\langle\underline{\pi}\langle x\rangle\rangle[x{\leftarrow}\langle\cdot\rangle] \\
\underline{(E :: [x{\leftarrow}\bar{t}], (y, \pi) :: D)} & := & \underline{(E, (y, \pi) :: D)}[x{\leftarrow}\bar{t}]
\end{array}
$$

The analysis of the Pointing WAM is based on a complex invariant that includes duality plus a generalization of the global closure invariant. We need an auxiliary definition:

Definition 9.2. *Given an environment E, we define its* slice $E{\upharpoonleft}$ *as the sequence of substitutions after the rightmost dumped substitution. Formally:*

$$
\begin{array}{rcl}
\epsilon{\upharpoonleft} & := & \epsilon \\
(E :: [x{\leftarrow}\bar{t}]){\upharpoonleft} & := & E{\upharpoonleft} :: [x{\leftarrow}\bar{t}] \\
(E :: [x{\leftarrow}\square]){\upharpoonleft} & := & \epsilon
\end{array}
$$

Moreover, if an environment E is of the form $E_1 :: [x{\leftarrow}\square] :: E_2$, we define $E{\upharpoonleft}_x := E_1{\upharpoonleft} :: [x{\leftarrow}\square] :: E_2$.

The notion of closed closure with global environment (Sect. 7.1) is extended to dummy constants $\square$ as expected.

Lemma 9.3 (Pointing WAM invariants)**.** *Let $s = \bar{t} \mid E \mid \pi \mid D$ be a Pointing WAM reachable state whose initial code $\bar{t}$ is well-named. Then:*

1. Subterm: *any code in s is a literal subterm of $\bar{t}$;*
2. Names: *the global closure of s is well-named.*
3. Dump-Environment Duality:
   - *(a) $(\underline{\pi}\langle\bar{t}\rangle, E{\upharpoonleft})$ is closed;*
   - *(b) for every pair (x, π') in D, $(\underline{\pi'}\langle x\rangle, E{\upharpoonleft}_x)$ is closed;*
   - *(c) $E \bot D$ holds.*
4. Contextual Decoding: *$\underline{(E, D)}$ is a call-by-need evaluation context.*

Proof. In the appendix, page 24. $\square$

The decoding of a state is defined as $\underline{\bar{t} \mid \pi \mid D \mid E} := \underline{(E, D)}\langle\underline{\pi}\langle\bar{t}\rangle\rangle$.

Theorem 9.4 (Pointing WAM Distillation)**.** $(\text{Pointing WAM}, \mathtt{Need}, \underline{\cdot})$ *is a reflective distillery. In particular, on a reachable state s we have:*

1. Commutative 1 & 2*: if $s \rightarrow_{c_1} s'$ or $s \rightarrow_{c_2} s'$ then $\underline{s} = \underline{s'}$;*
2. Multiplicative 1 & 2*: if $s \rightarrow_{m_1} s'$ or $s \rightarrow_{m_2} s'$ then $\underline{s} \multimap_{\mathtt{m}}\equiv \underline{s'}$;*

3. Exponential*: if* $s \to_e s'$ *then* $\underline{s} \multimap_{\mathtt{e}} =_\alpha \underline{s'}$*;*

Proof. Properties of the decoding:

1. *Conmutative 1*. We have
$$\underline{\overline{t}\,\overline{u} \mid \pi \mid D \mid E} \quad = \quad \underline{(E, D)}\langle\underline{\pi}\langle\overline{t}\,\overline{u}\rangle\rangle \quad = \quad \underline{\overline{t} \mid \overline{u} :: \pi \mid D \mid E}$$
2. *Conmutative 2*. Note that E_2 has no dumped substitutions, since $E_1 :: [x{\leftarrow}\Box] :: E_2 \bot (x, \pi) :: D$. Then:
$$\begin{array}{ll} \underline{x \mid \pi \mid D \mid E_1 :: [x{\leftarrow}\overline{t}] :: E_2} & = \\ \underline{e_2}\langle\overline{(E_1, D)}\langle\underline{\pi}\langle x\rangle\rangle[x{\leftarrow}\overline{t}]\rangle & = \\ \underline{\overline{t} \mid \epsilon \mid (x, \pi) :: D \mid E_1 :: [x{\leftarrow}\Box] :: E_2} & \end{array}$$
3. *Multiplicative, empty dump*.
$$\begin{array}{lcll} \underline{\lambda x.\overline{t} \mid \overline{u} :: \pi \mid \epsilon \mid E} & = & \underline{e}\langle\underline{\pi}\langle(\lambda x.\overline{t})\,\overline{u}\rangle\rangle & \multimap_{\mathtt{m}} \\ & & \underline{e}\langle\underline{\pi}\langle\overline{t}[x{\leftarrow}\overline{u}]\rangle\rangle & \equiv^*_{@l} \quad Lem.2.4 \\ & & \underline{e}\langle\underline{\pi}\langle\overline{t}\rangle[x{\leftarrow}\overline{u}]\rangle & = \\ & & \underline{\overline{t} \mid \pi \mid \epsilon \mid [x{\leftarrow}\overline{u}] :: E} & \end{array}$$
4. *Multiplicative, non-empty dump*.
$$\begin{array}{ll} \underline{\lambda x.\overline{t} \mid \overline{u} :: \pi \mid (y, \pi') :: D \mid E_1 :: [y{\leftarrow}\Box] :: E_2} & = \\ \underline{e_2}\langle\overline{(E_1, D)}\langle\underline{\pi'}\langle y\rangle\rangle[y{\leftarrow}\underline{\pi}\langle(\lambda x.\overline{t})\,\overline{u}\rangle]\rangle & \multimap_{\mathtt{m}} \\ \underline{e_2}\langle\overline{(E_1, D)}\langle\underline{\pi'}\langle y\rangle\rangle[y{\leftarrow}\underline{\pi}\langle\overline{t}[x{\leftarrow}\overline{u}]\rangle]\rangle & \equiv_{Lem.2.4} \\ \underline{e_2}\langle\overline{(E_1, D)}\langle\underline{\pi'}\langle y\rangle\rangle[y{\leftarrow}\underline{\pi}\langle\overline{t}\rangle][x{\leftarrow}\overline{u}]\rangle & = \\ \underline{\overline{t} \mid \pi \mid (y, \pi') :: D \mid E_1 :: [y{\leftarrow}\Box] :: [x{\leftarrow}\overline{u}] :: E_2} & \end{array}$$
5. *Exponential*.
$$\begin{array}{ll} \underline{\overline{v} \mid \epsilon \mid (x, \pi) :: D \mid E_1 :: [x{\leftarrow}\Box] :: E_2} & = \\ \underline{e_2}\langle\overline{(E_1, D)}\langle\underline{\pi}\langle x\rangle\rangle[x{\leftarrow}v]\rangle & \multimap_{\mathtt{e}} \\ \underline{e_2}\langle\overline{(E_1, D)}\langle\underline{\pi}\langle v\rangle\rangle[x{\leftarrow}v]\rangle & =_\alpha \\ \underline{e_2}\langle\overline{(E_1, D)}\langle\underline{\pi}\langle v^\alpha\rangle\rangle[x{\leftarrow}v]\rangle & = \\ \underline{\overline{v}^\alpha \mid \pi \mid D \mid E_1 :: [x{\leftarrow}\overline{v}] :: E_2} & \end{array}$$

Progress. Let $s = \overline{t} \mid \pi \mid D \mid E$ be a commutative normal form s.t. $\underline{s} \multimap u$. If $\overline{t}$ is

- *an application* $\overline{uw}$. Then a $\to_{c_1}$ transition applies and s is not a commutative normal form, absurd.
- *a variable* x. By the machine invariant, x must be bound by $E\!\upharpoonright$. So $E = E_1 :: [x{\leftarrow}\overline{u}] :: E_2$, a $\to_{c_2}$ transition applies, and s is not a commutative normal form, absurd.
- *an abstraction* $\overline{v}$. Two cases:
  - *The stack* π *is empty*. The dump D cannot be empty, since if $D = \epsilon$ we have that $\underline{s} = \underline{e}\langle\overline{v}\rangle$ is normal. So $D = (x, \pi') :: D'$. By duality, $E = E_1 :: [x{\leftarrow}\Box] :: E_2$ and a $\to_e$ transition applies.
  - *The stack* π *is non-empty*. If the dump D is empty, the first case of $\to_m$ applies. If $D = (x, \pi') :: D'$, by duality $E = E_1 :: [x{\leftarrow}\Box] :: E_2$ and the second case of $\to_m$ applies. □

10. Distillation Preserves Complexity

Here, for every abstract machine we bound the number of commutative steps $|\rho|_c$ in an execution ρ in terms of

1. the number of principal steps $|\rho|_p$,
2. the size $|\overline{t}|$ of the initial code $\overline{t}$.

The analysis only concerns the machines, but via the distillation theorems it expresses the length of the machine executions as a linear function of the length of the distilled derivations in the calculi. For every distillery, we will prove that the relationship is linear in both parameters, namely $|\rho|_c = O(|\overline{t}| \cdot |\rho|_p)$ holds.

Definition 10.1. *Let* $\mathtt{M}$ *be a distilled abstract machine and* $\rho : s \to^* s'$ *be an execution of initial code* $\overline{t}$*.* $\mathtt{M}$ *is*

1. ***Globally bilinear*** *if* $|\rho|_c = O((|\overline{t}| + 1) \cdot |\rho|_p)$*.*
2. ***Locally linear*** *if whenever* $s' \to^k_c s''$ *then* $k = O(|\overline{t}|)$*.*

The next lemma shows that local linearity is a sufficient condition for global bilinearity.

Proposition 10.2 (Locally Linear ⇒ Globally Bilinear)**.** *Let* $\mathtt{M}$ *be a locally linear distilled abstract machine, and* ρ *an execution of initial code* $\overline{t}$*. Then* $\mathtt{M}$ *is globally bilinear.*

Proof. The execution ρ writes uniquely as $\to^{k_1}_c \to^{h_1}_p \ldots \to^{k_m}_c \to^{h_m}_p$. By hypothesis $k_i = O(|\overline{t}|)$ for every $i \in \{1, \ldots, m\}$. From $m \le |\rho|_p$ follows that $|\rho|_c = O(|\overline{t}| \cdot |\rho|_p)$. We conclude with $|\rho| = |\rho|_p + |\rho|_c = |\rho|_p + O(|\overline{t}| \cdot |\rho|_p) = O((|\overline{t}| + 1) \cdot |\rho|_p)$. □

Call-by-name and call-by-value machines are easily seen to be locally linear, and thus globally bilinear.

Theorem 10.3. *KAM, MAM, CEK, LAM, and the Split CEK are locally linear, and so also globally bilinear.*

Proof. 1. *KAM/MAM*. Immediate: $\to_c$ reduces the size of the code, that is bounded by $|\overline{t}|$ by the subterm invariant.
2. *CEK*. Consider the following measure for states:
$$\#(\overline{u} \mid e \mid \pi) := \begin{cases} |\overline{u}| + |\overline{w}| & \text{if } \pi = \mathbf{a}(\overline{w}, e') :: \pi' \\ |\overline{u}| & \text{otherwise} \end{cases}$$
By direct inspection of the rules, it can be seen that both $\to_{c_1}$ and $\to_{c_2}$ transitions decrease the value of $\#$ for CEK states, and so the relation $\to_{c_1} \cup \to_{c_2}$ terminates (on reachable states). Moreover, both $|\overline{u}|$ and $|\overline{w}|$ are bounded by $|\overline{t}|$ by the subterm invariant (Lemma 6.1.2), and so $k \le 2 \cdot |\overline{t}| = O(|\overline{t}|)$.
3. *LAM* and *Split CEK*. Minor variations over the CEK, see the appendix (page 25) □

Call-by-need machines are not locally linear, because a sequence of $\to_{c_2}$ steps (remember $\to_c := \to_{c_1} \cup \to_{c_2}$) can be as long as the environment e, that is not bound by $|\overline{t}|$ (as for the MAM). Luckily, being locally linear is not a necessary condition for global bilinearity. We are in fact going to show that call-by-need machines are globally bilinear. The key observation is that $|\rho|_{c_2}$ is not only locally but also globally bound by $|\rho|_p$, as the next lemma formalizes.

We treat the WAM. The reasoning for the Merged WAM and for the Pointing WAM is analogous. Define $|\epsilon| := 0$ and $|(E, x, \pi) :: D| := 1 + |D|$. We have:

Lemma 10.4. *Let* $s = \overline{t} \mid \pi \mid D \mid E$ *be a WAM state, reached by the execution* ρ*. Then*

1. $|\rho|_{c_2} = |\rho|_e + |D|$.
2. $|E| + |D| \le |\rho|_m$
3. $|\rho|_{c_2} \le |\rho|_e + |\rho|_m = |\rho|_p$

Proof. 1. Immediate, as $\to_{c_2}$ is the only transition that pushes elements on D and $\to_e$ is the only transition that pops them.
2. The only rule that produces substitutions is $\to_m$. Note that 1) $\to_{c_2}$ and $\to_e$ preserve the global number of substitutions in a state; 2) e and D are made out of substitutions, if one considers every entry (e, x, π) of the dump as a substitution on x (and so the statement follows); 3) the inequality is given by the fact that an entry of the dump stocks an environment (counting for many substitutions).
3. Substitute Point 2 in Point 1. □

Theorem 10.5. *The WAM has globally linear commutations.*

Proof. Let ρ be an execution of initial code $\bar{t}$. Define $\rightarrow_{\neg c_1} := \rightarrow_e \cup \rightarrow_m \cup \rightarrow_{c_2}$ and note $|\rho|_{\neg c_1}$ the number of its steps in ρ. We estimate $\rightarrow_c := \rightarrow_{c_1} \cup \rightarrow_{c_2}$ by studying its components separately. For $\rightarrow_{c_2}$, Lemma 10.4.3 proves $|\rho|_{c_2} \leq |\rho|_p = O(|\rho|_p)$. For $\rightarrow_{c_1}$, as for the KAM, the length of a maximal $\rightarrow_{c_1}$ subsequence of ρ is bounded by $|\bar{t}|$. The number of $\rightarrow_{c_1}$ maximal subsequences of ρ is bounded by $|\rho|_{\neg c_1}$, that by Lemma 10.4.3 is linear in $O(|\rho|_p)$. Then $|\rho|_{c_1} = O(|\bar{t}| \cdot |\rho|_p)$. Summing up,

$$|\rho|_{c_2} + |\rho|_{c_1} = O(|\rho|_p) + O(|\bar{t}| \cdot |\rho|_p) = O((|\bar{t}| + 1) \cdot |\rho|_p) \qquad \square$$

11. Conclusions

The novelty of our study consists in using the linear substitution calculus (LSC) to discriminate between abstract machine transitions: some of them — the principal ones — are simulated, and thus shown to be logically relevant, while the others — the commutative ones — are sent on the structural congruence and have to be considered as bookkeeping operations. On one hand, the LSC is a sharp tool to study abstract machines. On the other hand, it provide an alternative framework which is *simpler* while being *conservative* at the level of complexity analysis.

Acknowledgments

A special acknowledgement to Claudio Sacerdoti Coen, for many useful discussions, comments and corrections to the paper. In particular, we owe him the intuition that a global analysis of call-by-need commutative rules may provide a linear bound. This work was partially supported by the ANR projects LOGOI (10-BLAN-0213-02) and COQUAS (ANR-12-JS02-006-01), by the French-Argentinian Laboratory in Computer Science **INFINIS**, the French-Argentinian project **ECOS-Sud** A12E04, the Qatar National Research Fund under grant NPRP 09-1107-1-168.

References

[1] B. Accattoli. An abstract factorization theorem for explicit substitutions. In *RTA*, pages 6–21, 2012.

[2] B. Accattoli. Evaluating functions as processes. In *TERMGRAPH*, pages 41–55, 2013.

[3] B. Accattoli and U. Dal Lago. On the invariance of the unitary cost model for head reduction. In *RTA*, pages 22–37, 2012.

[4] B. Accattoli and D. Kesner. The structural λ-calculus. In *CSL*, pages 381–395, 2010.

[5] B. Accattoli and L. Paolini. Call-by-value solvability, revisited. In *FLOPS*, pages 4–16, 2012.

[6] B. Accattoli, E. Bonelli, D. Kesner, and C. Lombardi. A nonstandard standardization theorem. In *POPL*, pages 659–670, 2014.

[7] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, pages 8–19, 2003.

[8] M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inf. Process. Lett.*, 90(5):223–232, 2004.

[9] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997.

[10] Z. M. Ariola, A. Bohannon, and A. Sabry. Sequent calculi and abstract machines. *ACM Trans. Program. Lang. Syst.*, 31(4), 2009.

[11] M. Biernacka and O. Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theor. Comput. Sci.*, 375(1-3):76–108, 2007.

[12] M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Log.*, 9(1), 2007.

[13] S. Chang and M. Felleisen. The call-by-need lambda calculus, revisited. In *ESOP*, pages 128–147, 2012.

[14] P. Clairambault. Estimation of the length of interactions in arena game semantics. In *FOSSACS*, pages 335–349, 2011.

[15] P. Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, 2007.

[16] V. Danos and L. Regnier. Head linear reduction. Technical report, 2004.

[17] V. Danos, H. Herbelin, and L. Regnier. Game semantics & abstract machines. In *LICS*, pages 394–405, 1996.

[18] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Technical Report RS-04-26, BRICS, 2004.

[19] O. Danvy and I. Zerny. A synthetic operational account of call-by-need evaluation. In *PPDP*, pages 97–108, 2013.

[20] N. G. de Bruijn. Generalizing Automath by Means of a Lambda-Typed Lambda Calculus. In *Mathematical Logic and Theoretical Computer Science*, number 106 in Lecture Notes in Pure and Applied Mathematics, pages 71–92. Marcel Dekker, 1987.

[21] R. Di Cosmo, D. Kesner, and E. Polonovski. Proof nets and explicit substitutions. *Math. Str. in Comput. Sci.*, 13(3):409–450, 2003.

[22] T. Ehrhard and L. Regnier. Böhm trees, Krivine's machine and the Taylor expansion of lambda-terms. In *CiE*, pages 186–197, 2006.

[23] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, Aug. 1986.

[24] R. Garcia, A. Lumsdaine, and A. Sabry. Lazy evaluation and delimited control. In *POPL*, pages 153–164, 2009.

[25] T. Hardin and L. Maranget. Functional runtime systems within the lambda-sigma calculus. *J. Funct. Program.*, 8(2):131–176, 1998.

[26] D. Kesner. A theory of explicit substitutions with safe and full composition. *Logical Methods in Computer Science*, 5(3), 2009.

[27] D. Kesner and S. Lengrand. Resource operators for lambda-calculus. *Inf. Comput.*, 205(4):419–473, 2007.

[28] D. Kesner and F. Renaud. The prismoid of resources. In *MFCS*, pages 464–476, 2009.

[29] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[30] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, Jan. 1964. . URL `http://dx.doi.org/10.1093/comjnl/6.4.308`.

[31] F. Lang. Explaining the lazy Krivine machine using explicit substitution and addresses. *Higher-Order and Symbolic Computation*, 20(3): 257–270, 2007.

[32] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990. URL `http://gallium.inria.fr/ xleroy/publi/ZINC.pdf`.

[33] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998.

[34] J. Maraist, M. Odersky, D. N. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theor. Comput. Sci.*, 228(1-2):175–210, 1999.

[35] G. Mascari and M. Pedicini. Head linear reduction and pure proof net extraction. *Theor. Comput. Sci.*, 135(1):111–137, 1994.

[36] R. Milner. Local bigraphs and confluence: Two conjectures. *Electr. Notes Theor. Comput. Sci.*, 175(3):65–73, 2007.

[37] R. P. Nederpelt. The fine-structure of lambda calculus. Technical Report CSN 92/07, Eindhoven Univ. of Technology, 1992.

[38] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[39] P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Program*, 7(3): 231–264, 1997.

A. Technical Appendix: proofs of the determinism of the calculi (Proposition 2.1)

A.1 Call-by-Name

Let $t = H_1\langle r_1\rangle = H_2\langle r_2\rangle$. By induction on the structure of t. Cases:

- **Variable or an abstraction.** Vacuously true, because there is no redex.
- **Application.** Let $t = uw$. Suppose that one of the two evaluation contexts, for instance H_1, is equal to $\langle\cdot\rangle$. Then, we must have $u = \lambda x.u'$, but in that case it is easy to see that the result holds, because H_2 cannot have its hole to the right of an application (in w) or under an abstraction (in u'). We may then assume that none of H_1, H_2 is equal to $\langle\cdot\rangle$. In that case, we must have $H_1 = H_1'w$ and $H_2 = H_2'w$, and we conclude by induction hypothesis.
- **Substitution.** Let $t = u[x{\leftarrow}w]$. This case is entirely analogous to the previous one.

A.2 Left-to-Right Call-by-Value

We prove the following statement, of which the determinism of the reduction is a consequence.

Lemma A.1. *Let t be a term. Then t has at most one subterm u that verifies both (i) and (ii):*

(i) Either u is a variable x, or u is an application $L\langle v\rangle L'\langle v'\rangle$, for v, v' being values.
(ii) u is under a left-to-right call-by-value evaluation context, i.e. $t = V\langle u\rangle$.

From the statement it follows that there is at most one $\multimap$-redex in t, *i.e.* $\multimap$ is deterministic.

Proof. by induction on the structure of t:

- t **is a variable.** There is only one subterm, under the empty evaluation context.
- t **is an abstraction.** There are no subterms that verify both (i) and (ii), since the only possible evaluation context is the empty one.
- t **is an application** $w\,r$**.** There are three possible situations:
  - *The left subterm w is not of the form $L\langle v\rangle$.* Then u cannot be at the root, *i.e.* $u \neq t$. Since $w\langle\cdot\rangle$ is not an evaluation context, u must be internal to $\langle\cdot\rangle r$, which is an evaluation context. We conclude by *i.h.*.
  - *The left subterm w is of the form $L\langle v\rangle$ with v a value, but the right subterm r is not.* Then u cannot be a subterm of w, and also $u \neq t$. Hence, if there is a subterm u as in the statement, it must be internal to the evaluation context $w\langle\cdot\rangle$. We conclude by *i.h.*.
  - *Both subterms have that form, i.e.* $w = L\langle v\rangle$ and $r = L'\langle v'\rangle$ with v and v' values. The only subterm that verifies both (i) and (ii) is $u = t$.
- t **is a substitution** $w[x{\leftarrow}r]$**.** Any occurrence of u must be internal to w (because $w[x{\leftarrow}\langle\cdot\rangle]$ is not an evaluation context). We conclude by *i.h.* that there is at most one such occurrence.

□

A.3 Right-to-Left Call-by-Value

Exactly as in the case for left-to-right call-by-value, we prove the following property, from which determinism of the reduction follows.

Lemma A.2. *Let t be a term. Then t has at most one subterm u that verifies both (i) and (ii):*

(i) u is either a variable x or an application $L\langle v\rangle L'\langle v'\rangle$, where v and v' are values.
(ii) u is under a right-to-left call-by-value evaluation context, i.e. $t = S\langle u\rangle$.

As a corollary, any term t has at most one $\multimap$-redex.

Proof. By induction on the structure of t:

- **Variable or abstraction.** Immediate.
- **Application.** If $t = w\,r$, there are three cases:
  - *The right subterm r is not of the form $L'\langle v'\rangle$.* Then u cannot be at the root. Since $\langle\cdot\rangle\,r$ is not an evaluation context, u must be internal to r and we conclude by *i.h.*.
  - *The right subterm r is of the form $L'\langle v'\rangle$ but the left subterm w is not.* Again u cannot be at the root. Moreover, r has no applications or variables under an evaluation context. Therefore u must be internal to w and we conclude by *i.h.*.
  - *Both subterms have that form, i.e.* $w = L\langle v\rangle$ and $r = L'\langle v'\rangle$. We first note that w and r have no applications or variables under an evaluation context. The only possibility that remains is that u is at the root, *i.e.* $u = t$.
- **Substitution.** If $t = w[x{\leftarrow}r]$ is a substitution, u must be internal to w (because $w[x{\leftarrow}\langle\cdot\rangle]$ is not an evaluation context), and we conclude by *i.h.*.

□

A.4 Call-by-Need

We first need an auxiliary result:

Lemma A.3. *Let $t := N\langle x\rangle$ for an evaluation context N such that $x \in \mathtt{fv}(t)$. Then:*

1. *for every substitution context L and abstraction v, $t \neq L\langle v\rangle$;*
2. *for every evaluation context N' and variable y, $t = N'\langle y\rangle$ implies $N' = N$ and $y = x$;*
3. *t is a call-by-need normal form.*

Proof. In all points we use a structural induction on N. For point 1:

- $N = \langle\cdot\rangle$: obvious.
- $N = N_1 u$: obvious.
- $N = N_1[y{\leftarrow}w]$: suppose that $L = L'[y{\leftarrow}w]$ (for otherwise the result is obvious); then we apply the induction hypothesis to N_1 to obtain $N_1\langle x\rangle \neq L'\langle v\rangle$.
- $N = N_1\langle y\rangle[y{\leftarrow}N_2]$: suppose that $L = L'[y{\leftarrow}N_2\langle x\rangle]$ (for otherwise the result is obvious); then we apply the induction hypothesis to N_1 to obtain $N_1\langle y\rangle \neq L'\langle v\rangle$.

For point 2:

- $N = \langle\cdot\rangle$: obvious.
- $N = N_1 u$: we must necessarily have $N' = N_1' u$ and we conclude by induction hypothesis.
- $N = N_1[z{\leftarrow}u]$: in principle, there are two cases. First, we may have $N' = N_1'[z{\leftarrow}u]$, which allows us to conclude immediately by induction hypothesis, as above. The second possibility would be $N' = N_1'\langle z\rangle[z{\leftarrow}N_2']$, with $N_2'\langle y\rangle = u$, but this is actually impossible. In fact, it would imply $N_1\langle x\rangle = N_1'\langle z\rangle$, which by induction hypothesis would give us $z = x$, contradicting the hypothesis $x \in \mathtt{fv}(t)$.
- $N = N_1\langle z\rangle[z{\leftarrow}N_2]$: by symmetry with the above case, the only possibility is $N' = N_1\langle z\rangle[z{\leftarrow}N_2']$, which allows us to conclude immediately by induction hypothesis.

For point 3, let r be a redex (*i.e.*, a term matching the left hand side of $\mapsto_{\mathtt{dB}}$ or $\mapsto_{\mathtt{lsv}}$) and let N' be an evaluation context. We will show by structural induction on N that $t \neq N'\langle r\rangle$. We will do this by considering, in each inductive case, all the possible shapes of N'.

- $N = \langle\cdot\rangle$: obvious.
- $N = N_1 u$: the result is obvious unless $N' = \langle\cdot\rangle$ or $N' = N_1' u$. In the latter case, we conclude by induction hypothesis (on N_1). In the former case, since r is a redex, we are forced to have $r = L\langle v\rangle u'$ for some abstraction v, substitution context L and term u'. Now, even supposing $u' = u$, we are still allowed to conclude because $N_1\langle x\rangle \neq L\langle v\rangle$ by point 1.
- $N = N_1[y{\leftarrow}u]$: the result is obvious unless:
  - $N' = \langle\cdot\rangle$: this time, the fact that r is a redex forces $r = N_1'\langle y\rangle[y{\leftarrow}u]$. Even if we admit that $u = L\langle v\rangle$, we may still conclude because $x \neq y$ (by the hypothesis $x \in \mathtt{fv}(t)$), hence $N_1\langle x\rangle \neq N_1'\langle y\rangle$ by point 2.
  - $N' = N_1'[y{\leftarrow}u]$: immediate by induction hypothesis on N_1.
  - $N' = N_1'\langle y\rangle[y{\leftarrow}N_2']$: even if $N_2'\langle r\rangle = u$, we may still conclude because, again, $x \neq y$ implies $N_1\langle x\rangle \neq N_1'\langle y\rangle$ by point 2.
- $N = N_1\langle y\rangle[y{\leftarrow}N_2]$: again, the result is obvious unless:
  - $N' = \langle\cdot\rangle$: the fact that r is a redex implies $r = N_1'\langle y\rangle[y{\leftarrow}L\langle v\rangle]$. Even assuming $N_1' = N_1$, we may still conclude because $N_2\langle x\rangle \neq L\langle v\rangle$ by point 1.
  - $N' = N_1'[y{\leftarrow}N_2\langle x\rangle]$: since $y \in \mathtt{fv}(N_1\langle y\rangle)$, we conclude because the induction hypothesis gives us $N_1\langle y\rangle \neq N_1'\langle r\rangle$.
  - $N' = N_1\langle y\rangle[y{\leftarrow}N_2']$: we conclude at once by applying the induction hypothesis to N_2.

□

Now, the proof of Proposition 2.1 is by structural induction on $t := N_1\langle r_1\rangle = N_2\langle r_2\rangle$. Cases:

- **Variable or abstraction.** Impossible, since variables and abstractions are both call-by-need normal.
- **Application, *i.e.* $t = uw$.** This case is treated exactly as in the corresponding case of the proof of Proposition 2.1.
- **Substitution, *i.e.* $t = u[x{\leftarrow}w]$.** Cases:
  - *Both contexts have their holes in u or w.* It follows from the *i.h.*.
  - *One of the contexts—say N_1—is empty, i.e. $u = N_3\langle x\rangle$, $w = L\langle v\rangle$, and $r_1 = N_3\langle x\rangle[x{\leftarrow}L\langle v\rangle]$.* This case is impossible. Indeed, 1) the hole of N_2 cannot be in $L\langle v\rangle$, because it is call-by-need normal, and 2) it cannot be inside $N_3\langle x\rangle$ because by Lemma A.3.3 $N\langle x\rangle$ is call-by-need normal.
  - *One of the contexts—say N_1—has its hole in w and the other one has its hole in u, i.e. $N_1 = N_3\langle x\rangle[x{\leftarrow}N_4]$ and $N_2 = N_5[x{\leftarrow}w]$.* This case is impossible, because by Lemma A.3.3 $N_3\langle x\rangle$ is call-by-need normal.

B. Technical Appendix: proofs of strong bisimulation

B.1 Proof of Proposition 2.2 ($\equiv$ is a strong bisimulation) for call-by-name

Before proving the main result, we need two auxiliary lemmas, proved by straightforward inductions on H:

Lemma B.1. *Let t be a term, H be a call-by-name evaluation context not capturing any variable in $\mathtt{fv}(t)$, and $x \notin \mathtt{fv}(H\langle y\rangle)$. Then $H\langle t[x{\leftarrow}u]\rangle \equiv H\langle t\rangle[x{\leftarrow}u]$.*

Lemma B.2. *The equivalence relation $\equiv$ as defined for call-by-name preserves the shape of $H\langle x\rangle$. More precisely, if $H\langle x\rangle \equiv t$, with x not captured by H, then t is of the form $H'\langle x\rangle$, with x not captured by H'.*

Now we turn to the proof of Proposition 2.2 itself.

Let $\Longleftrightarrow$ be the symmetric closure of the union of the axioms defining $\equiv$ for call-by-name, that is of $\equiv_{gc} \cup \equiv_{dup} \cup \equiv_{@} \cup \equiv_{com} \cup \equiv_{[\cdot]}$. Note that $\equiv$ is the reflexive–transitive closure of $\Longleftrightarrow$. The proof is in two parts:

(I) Prove the property holds for $\Longleftrightarrow$, *i.e.* if $t \multimap_a u$ and $t \Longleftrightarrow w$, there exists r s.t. $w \multimap_a r$ and $u \equiv r$.
(II) Prove the property holds for $\equiv$ (*i.e.* for many steps of $\Longleftrightarrow$) by resorting to (I).

The proof of (II) is immediate by induction on the number of $\Longleftrightarrow$ steps. The proof of (I) goes by induction on the rewriting step $\multimap$ (that, since $\multimap$ is closed by evaluation contexts, becomes a proof by induction on the evaluation context H). In principle, we should always consider the two directions of $\Longleftrightarrow$. Most of the time, however, one direction is obtained by simply reading the diagram of the other direction bottom-up, instead than top-down; these cases are simply omitted, we distinguish the two directions only when it is relevant.

1. **Base case 1: multiplicative root step $t = L\langle\lambda x.t'\rangle u' \mapsto_{\mathtt{dB}} L\langle t'[x{\leftarrow}u']\rangle = u$.**
   If the $\Longleftrightarrow$ step is internal to u' or internal to one of the substitutions in L, the pattern of the $\Longleftrightarrow$ redex does not overlap with the $\mapsto_{\mathtt{dB}}$ step, and the proof is immediate, the two steps commute. Otherwise, we consider every possible case for $\Longleftrightarrow$:
   (a) *Garbage Collection* $\equiv_{gc}$. The garbage collected substitution must be one of the substitutions in L, *i.e.* L must be of the form $L'\langle L''[y{\leftarrow}w']\rangle$. Let $\widehat{L} := L'\langle L''\rangle$. Then:
   $$\begin{array}{ccc} L\langle\lambda x.t'\rangle u' & \xrightarrow{\mathtt{dB}} & L\langle t'[x{\leftarrow}u']\rangle \\ \equiv_{gc} & & \equiv_{gc} \\ \widehat{L}\langle\lambda x.t'\rangle u' & \xrightarrow{\mathtt{dB}} & \widehat{L}\langle t'[x{\leftarrow}u']\rangle \end{array}$$
   (b) *Duplication* $\equiv_{dup}$. The duplicated substitution must be one of the substitutions in L, *i.e.* L must be of the form $L'\langle L''[y{\leftarrow}w']\rangle$. Then:
   $$\begin{array}{ccc} L'\langle L''\langle\lambda x.t'\rangle[y{\leftarrow}w']\rangle u' & \xrightarrow{\mathtt{dB}} & t_1 \\ \equiv_{dup} & & \equiv_{dup} \\ t_2 & \xrightarrow{\mathtt{dB}} & t_3 \end{array}$$
   where
   $$t_1 := L'\langle L''\langle t'[x{\leftarrow}u']\rangle[y{\leftarrow}w']\rangle,$$
   $$t_2 := L'\langle\ (L''\langle\lambda x.t'\rangle)_{[z]_y}[y{\leftarrow}w'][z{\leftarrow}w']\ \rangle u',$$
   $$t_3 := L'\langle\ (L''\langle t'[x{\leftarrow}u']\rangle)_{[z]_y}[y{\leftarrow}w'][z{\leftarrow}w']\ \rangle.$$
   (c) *Commutation with application* $\equiv_{@}$. Here $\equiv_{@}$ can only be applied in one direction. The diagram is:

$$
\begin{tikzcd}
L\langle\lambda y.t'\rangle[x\leftarrow q']u'[x\leftarrow q'] \arrow[d, "\mathtt{dB}"] \arrow[r, phantom, "\equiv_{@}"] & t_1 \arrow[d, dashed, "\mathtt{dB}"] \\
t_4 \arrow[dd, phantom, "\equiv_{\alpha}"] & t_2 \arrow[d, phantom, "\equiv_{dup}"] \\
& t_3 \arrow[d, phantom, "\equiv_{com}^{*}"] \\
t_5 \arrow[r, phantom, "\equiv_{[\cdot]}"] & t_6
\end{tikzcd}
$$

where

$$
\begin{aligned}
t_1 &:= (L\langle\lambda y.t'\rangle u')[x\leftarrow q'],\\
t_2 &:= (L\langle t'[y\leftarrow u']\rangle)[x\leftarrow q'],\\
t_3 &:= (L\langle t'[y\leftarrow u'\{x\leftarrow y\}]\rangle)[x\leftarrow q'][y\leftarrow q'],\\
t_4 &:= L\langle t'[y\leftarrow u'[x\leftarrow q']]\rangle[x\leftarrow q'],\\
t_5 &:= (L\langle t'[y\leftarrow u'\{x\leftarrow y\}[y\leftarrow q']]\rangle)[x\leftarrow q'],\\
t_6 &:= (L\langle t'[y\leftarrow u'\{x\leftarrow y\}][y\leftarrow q']\rangle)[x\leftarrow q'].
\end{aligned}
$$

(d) *Commutation of independent substitutions* $\equiv_{com}$. The substitutions that are commuted by the $\equiv_{com}$ rule must be both in L, *i.e.* L must be of the form $L'\langle L''[y\leftarrow w'][z\leftarrow r']\rangle$ with $z \notin \mathtt{fv}(w')$. Let $\widehat{L} = L'\langle L''[z\leftarrow r'][y\leftarrow w']\rangle$. Then:

$$
\begin{tikzcd}
L\langle\lambda x.t'\rangle u' \arrow[r, "\mathtt{dB}"] \arrow[d, phantom, "\equiv_{com}"] & L\langle t'[x\leftarrow u']\rangle \arrow[d, phantom, "\equiv_{com}"] \\
\widehat{L}\langle\lambda x.t'\rangle u' \arrow[r, dashed, "\mathtt{dB}"] & \widehat{L}\langle t'[x\leftarrow u']\rangle
\end{tikzcd}
$$

(e) *Composition of substitutions* $\equiv_{[\cdot]}$. The substitutions that appear in the left-hand side of the $\equiv_{[\cdot]}$ rule must both be in L, *i.e.* L must be of the form $L'\langle L''[y\leftarrow w'][z\leftarrow r']\rangle$ with $z \notin \mathtt{fv}(L''\langle\lambda x.t'\rangle)$. Let $\widehat{L} = L'\langle L''[y\leftarrow w'[z\leftarrow r']]\rangle$. Exactly as in the previous case:

$$
\begin{tikzcd}
L\langle\lambda x.t'\rangle u' \arrow[r, "\mathtt{dB}"] \arrow[d, phantom, "\equiv_{[\cdot]}"] & L\langle t'[x\leftarrow u']\rangle \arrow[d, phantom, "\equiv_{[\cdot]}"] \\
\widehat{L}\langle\lambda x.t'\rangle u' \arrow[r, dashed, "\mathtt{dB}"] & \widehat{L}\langle t'[x\leftarrow u']\rangle
\end{tikzcd}
$$

2. **Base case 2: exponential root step** $t = H'\langle x\rangle[x\leftarrow t'] \mapsto_{\mathtt{ls}} H'\langle t'\rangle[x\leftarrow t'] = u$**.** If the $\Longleftrightarrow$ step is internal to t', the proof is immediate, since there is no overlap with the pattern of the $\mapsto_{\mathtt{ls}}$ redex. Similarly, if the $\Longleftrightarrow$ step is internal to $H\langle x\rangle$, the proof is straightforward by resorting to Lemma B.2.
Now we proceed by case analysis on the $\Longleftrightarrow$ step:

(a) *Garbage collection* $\equiv_{gc}$. Note that $\equiv_{gc}$ cannot remove $[x\leftarrow t']$, because by hypothesis x does occur in its scope. If the removed substitution belongs to H', *i.e.* $H' = H''\langle H'''[y\leftarrow u']\rangle$. Let $\widehat{H'} := H''\langle H'''\rangle$. Then:

$$
\begin{tikzcd}
H'\langle x\rangle[x\leftarrow t'] \arrow[r, "\mathtt{ls}"] \arrow[d, phantom, "\equiv_{gc}"] & H'\langle t'\rangle[x\leftarrow t'] \arrow[d, phantom, "\equiv_{gc}"] \\
\widehat{H'}\langle x\rangle[x\leftarrow t'] \arrow[r, dashed, "\mathtt{ls}"] & \widehat{H'}\langle t'\rangle[x\leftarrow t']
\end{tikzcd}
$$

If $\equiv_{gc}$ adds a substitution as topmost constructor the diagram is analogous.

(b) *Duplication* $\equiv_{dup}$. Two sub-cases:

i. *The equivalence $\equiv_{dup}$ acts on a substitution internal to H'.* This case goes as for Garbage collection.

ii. *The equivalence $\equiv_{dup}$ acts on $[x\leftarrow t']$.* There are two further sub-cases:

- *The substituted occurrence is renamed by $\equiv_{dup}$:*

$$
\begin{tikzcd}
H'\langle x\rangle[x\leftarrow t'] \arrow[r, "\mathtt{ls}"] \arrow[d, phantom, "\equiv_{gc}"] & H'\langle t'\rangle[x\leftarrow t'] \arrow[d, phantom, "\equiv_{gc}"] \\
H'_{[y]_x}\langle y\rangle[x\leftarrow t'][y\leftarrow t'] \arrow[r, dashed, "\mathtt{ls}"] & t_1
\end{tikzcd}
$$

where $t_1 := H'_{[y]_x}\langle t'\rangle[x\leftarrow t'][y\leftarrow t']$ and $H'_{[y]_x}$ is the context obtained from H' by renaming some (possibly none) occurrences of x as y.

- *The substituted occurrence is not renamed by $\equiv_{dup}$.* Essentially as in the previous case:

$$
\begin{tikzcd}
H'\langle x\rangle[x\leftarrow t'] \arrow[r, "\mathtt{ls}"] \arrow[d, phantom, "\equiv_{dup}"] & H'\langle t'\rangle[x\leftarrow t'] \arrow[d, phantom, "\equiv_{dup}"] \\
H'_{[y]_x}\langle x\rangle[x\leftarrow t'][y\leftarrow t'] \arrow[r, dashed, "\mathtt{ls}"] & t_1
\end{tikzcd}
$$

where $t_1 := H'_{[y]_x}\langle t'\rangle[x\leftarrow t'][y\leftarrow t']$.

(c) *Commutation with application* $\equiv_{@}$. Two sub-cases:

i. *The equivalence $\equiv_{@}$ acts on a substitution internal to H'.* This case goes as for Garbage collection.

ii. *The equivalence $\equiv_{@}$ acts on $[x\leftarrow t']$.* It must be the case that H' is of the form $H''u'$. Then:

$$
\begin{tikzcd}
(H''\langle x\rangle u')[x\leftarrow t'] \arrow[r, "\mathtt{ls}"] \arrow[d, phantom, "\equiv_{@}"] & t_1 \arrow[d, phantom, "\equiv_{@}"] \\
t_2 \arrow[r, dashed, "\mathtt{ls}"] & t_3
\end{tikzcd}
$$

where

$$
\begin{aligned}
t_1 &:= (H''\langle t'\rangle u')[x\leftarrow t'],\\
t_2 &:= H''\langle x\rangle[x\leftarrow t']u'[x\leftarrow t'],\\
t_3 &:= H''\langle t'\rangle[x\leftarrow t']u'[x\leftarrow t'].
\end{aligned}
$$

(d) *Commutation of independent substitutions* $\equiv_{com}$. Two sub-cases:

i. *The equivalence $\equiv_{com}$ acts on two substitutions internal to H'.* This case goes as for Garbage collection.

ii. *The equivalence $\equiv_{com}$ acts on $[x\leftarrow t']$.* It must be the case that H' is of the form H''. Then:

$$
\begin{tikzcd}
H''\langle x\rangle[y\leftarrow u'][x\leftarrow t'] \arrow[r, "\mathtt{ls}"] \arrow[d, phantom, "\equiv_{com}"] & H''\langle t'\rangle[y\leftarrow u'][x\leftarrow t'] \arrow[d, phantom, "\equiv_{com}"] \\
H''\langle x\rangle[x\leftarrow t'][y\leftarrow u'] \arrow[r, dashed, "\mathtt{ls}"] & H''\langle t'\rangle[x\leftarrow t'][y\leftarrow u']
\end{tikzcd}
$$

(e) *Composition of substitutions* $\equiv_{[\cdot]}$. Two sub-cases:

i. *The equivalence $\equiv_{[\cdot]}$ acts on two substitutions internal to H'.* This case goes as for Garbage collection.

ii. *The equivalence $\equiv_{[\cdot]}$ acts on $[x\leftarrow t']$.* Note that the equivalence $\equiv_{[\cdot]}$ cannot be applied from left to right to $[x\leftarrow t']$, because $H'\langle x\rangle$ must be of the form $H''\langle x\rangle[y\leftarrow u']$ with $x \notin \mathtt{fv}(H''\langle x\rangle)$, which is clearly not possible. It can be applied from right to left. The diagram is:

$$\begin{tikzcd}
H'\langle x\rangle[x\leftarrow t'[y\leftarrow u]] \arrow[rr, phantom, "\equiv_{@}"] \arrow[d, "\mathtt{ls}"] & & t_1 \arrow[d, dashed, "\mathtt{ls}"] \\
t_4 \arrow[d, phantom, "\equiv \text{ by Lemma B.1}"] & & t_2 \arrow[d, phantom, "\equiv_{dup}"] \\
t_5 \arrow[d, phantom, "\equiv_{[\cdot]}"] & & t_3 \arrow[d, phantom, "\equiv_{com}"] \\
t_6 \arrow[rr, phantom, "\equiv_{\alpha}"] & & t_7
\end{tikzcd}$$

where

$$t_1 := H'\langle x\rangle[x\leftarrow t'][y\leftarrow u],$$
$$t_2 := H'\langle t'\rangle[x\leftarrow t'][y\leftarrow u],$$
$$t_3 := H'\langle t'\{y\leftarrow z\}\rangle[x\leftarrow t'][z\leftarrow u][y\leftarrow u],$$
$$t_4 := H'\langle t'[y\leftarrow u]\rangle[x\leftarrow t'[y\leftarrow u]],$$
$$t_5 := H'\langle t'\rangle[y\leftarrow u][x\leftarrow t'[y\leftarrow u]],$$
$$t_6 := H'\langle t'\rangle[y\leftarrow u][x\leftarrow t'][y\leftarrow u],$$
$$t_7 := H'\langle t'\{y\leftarrow z\}\rangle[z\leftarrow u][x\leftarrow t'][y\leftarrow u].$$

3. **Inductive case 1: left of an application** $H = H'q$**.** The situation is:
$$t = t'q \multimap_a u'q = u$$
for terms t', u' such that either $t' \multimap_{\mathtt{m}} u'$ or $t' \multimap_{\mathtt{e}} u'$. Two sub-cases:
   (a) *The* $t \Longleftrightarrow w$ *step is internal to* t'. The proof simply uses the *i.h.* applied to the (strictly smaller) evaluation context of the step $t' \multimap_a u'$.
   (b) *The* $t \Longleftrightarrow w$ *step involves the topmost application*. The $\Longleftrightarrow$ step can only be a commutation with the root application. Moreover, for $t'q$ to match with the right-hand side of the $\equiv_{@}$ rule, t' must have the form $w'[x\leftarrow r']$ and q the form $q'[x\leftarrow r']$, so that the $\Longleftrightarrow$ is:
$$w = (w'q')[x\leftarrow r'] \equiv_{@} w'[x\leftarrow r']q'[x\leftarrow r'] = t$$
   Three sub-cases:
      i. *The rewriting step is internal to* w'. Then the two steps trivially commute. Let $a \in \{\mathtt{dB}, \mathtt{ls}\}$:
$$\begin{tikzcd}
w'[x\leftarrow r']q'[x\leftarrow r'] \arrow[r, "a"] \arrow[d, phantom, "\equiv_{@}"] & w''[x\leftarrow r']q'[x\leftarrow r'] \arrow[d, phantom, "\equiv_{@}"] \\
(w'q')[x\leftarrow r'] \arrow[r, dashed, "a"] & (w''q')[x\leftarrow r']
\end{tikzcd}$$
      ii. $\mathtt{dB}$*-step not internal to* w'. Exactly as the multiplicative root case 1c (read in the other direction).
   (c) $\mathtt{ls}$*-step not internal to* w'. Not possible: the topmost constructor is an application, consequently any $\multimap_{\mathtt{e}}$ has to take place in w'.
4. **Inductive case 2: left of a substitution** $H = H'[x\leftarrow q]$**.** The situation is:
$$t = t'[x\leftarrow q] \multimap u'[x\leftarrow q] = u$$
with $t' = H'\langle t''\rangle$. If the $\Longleftrightarrow$ step is internal to $H'\langle t'\rangle$, the proof we conclude using the *i.h.*. Otherwise:
   (a) *Garbage Collection* $\equiv_{gc}$. If the garbage collected substitution is $[x\leftarrow q]$ then:
$$\begin{tikzcd}
t'[x\leftarrow q] \arrow[r] \arrow[d, phantom, "\equiv_{gc}"] & u'[x\leftarrow q] \arrow[d, phantom, "\equiv_{gc}"] \\
t' \arrow[r, dashed] & u'
\end{tikzcd}$$
   If the substitution is introduced out of the blue, *i.e.* $t'[x\leftarrow q] \equiv_{gc} t'[x\leftarrow q][y\leftarrow q']$ or $t'[x\leftarrow q] \equiv_{gc} t'[y\leftarrow q'][x\leftarrow q]$ the diagram is analogous.
   (b) *Duplication* $\equiv_{dup}$. If the duplicated substitution is $[x\leftarrow q]$ then:
$$\begin{tikzcd}
t'[x\leftarrow q] \arrow[r] \arrow[d, phantom, "\equiv_{dup}"] & u'[x\leftarrow q] \arrow[d, phantom, "\equiv_{dup}"] \\
t'_{[y]_x}[x\leftarrow q][y\leftarrow q] \arrow[r, dashed] & u'_{[y]_x}[x\leftarrow q]
\end{tikzcd}$$
   If duplication is applied in the other direction, *i.e.* $t' = t''[y\leftarrow q]$ and
$$t'[x\leftarrow q] = t''[y\leftarrow q][x\leftarrow q] \equiv_{dup} t''\{y\leftarrow x\}[x\leftarrow q] = t'[x\leftarrow q]$$
   the interesting case is when $t'' = H''\langle y\rangle$ and the step is exponential:
$$\begin{tikzcd}
H''\langle y\rangle[y\leftarrow q][x\leftarrow q] \arrow[r, "\mathtt{ls}"] \arrow[d, phantom, "\equiv_{dup}"] & H''\langle q\rangle[y\leftarrow q][x\leftarrow q] \arrow[d, phantom, "\equiv_{dup}"] \\
H''\langle x\rangle\{y\leftarrow x\}[x\leftarrow q] \arrow[r, dashed, "\mathtt{ls}"] & H''\langle q\rangle\{y\leftarrow x\}[x\leftarrow q]
\end{tikzcd}$$
   If t' is $H''\langle x\rangle$ it is an already treated base case and if t' has another form the rewriting step does not interact with the duplication, and so they simply commute.
   (c) *Commutation with application* $\equiv_{@}$. Then $t' = t''u''$. Three sub-cases:
      i. *The* $\multimap$ *step is internal to* t''. Then:
$$\begin{tikzcd}
(t''u'')[x\leftarrow q] \arrow[r] \arrow[d, phantom, "\equiv_{@}"] & (t'''u'')[x\leftarrow q] \arrow[d, phantom, "\equiv_{@}"] \\
t''[x\leftarrow q]u''[x\leftarrow q] \arrow[r, dashed] & t'''[x\leftarrow q]u''[x\leftarrow q]
\end{tikzcd}$$
      ii. *The* $\multimap$ *step is a multiplicative step*. If $t'' = L\langle\lambda y.t'''\rangle$ then it goes like the diagram of the multiplicative root case 1c (read in the other direction).
      iii. *The* $\multimap$ *step is an exponential step*. Then it must be $[x\leftarrow q]$ that substitutes on the head variable, but this case has already been treated as a base case (case 2c).
   (d) *Commutation of independent substitutions* $\equiv_{com}$. It must be $t' = t''[y\leftarrow q']$ with $x \notin \mathtt{fv}(q')$, so that $t''[y\leftarrow q'][x\leftarrow q] \equiv_{com} t''[x\leftarrow q][y\leftarrow q']$. Three sub-cases:
      i. *Reduction takes place in* t''. Then reduction and the equivalence simply commute, as in case 4(c)i.
      ii. *Exponential steps involving* $[x\leftarrow q]$. This is an already treated base case (case 2(d)ii).
      iii. *Exponential step involving* $[y\leftarrow q']$. This case is solved reading bottom-up the diagram of case 2(d)ii.
   (e) *Composition of substitutions* $\equiv_{[\cdot]}$. It must be $t' = t''[y\leftarrow q']$ with $x \notin \mathtt{fv}(t'')$, so that $t''[y\leftarrow q'][x\leftarrow q] \equiv_{[\cdot]} t''[x\leftarrow q[y\leftarrow q']]$. Three sub-cases:
      i. *Reduction takes place in* t''. Then reduction and the equivalence simply commute, as in case 4(c)i.
      ii. *Exponential steps involving* $[x\leftarrow q]$. This case is solved reading bottom-up the diagram of case 2(e)ii.
      iii. *Exponential step involving* $[y\leftarrow q']$. Impossible, because by hypothesis $x \notin \mathtt{fv}(t'')$.

B.2 Proof of Proposition 2.2 ($\equiv$ is a strong bisimulation) for left-to-right call-by-value

We follow the structure of the proof in Sect. B.1 for call-by-name. Structural equivalence for call-by-value is defined exactly in the same way.

Before proving the main result, we need the following auxiliary lemmas, proved by straightforward inductions on the contexts. Lemma B.3.2 is the adaptation of Lemma B.2 already stated for call-by-name:

Lemma B.3. *The equivalence relation $\equiv$ preserves the "shapes" of $L\langle v\rangle$ and $V\langle x\rangle$. Formally:*

1. *If $L\langle v\rangle \equiv t$, then t is of the form $L'\langle v'\rangle$.*
2. *If $V\langle x\rangle \equiv t$, with x not bound by V, then t is of the form $V'\langle x\rangle$, with x not bound by V'.*

Lemma B.4. $L\langle t[x{\leftarrow}u]\rangle \equiv L\langle t[x{\leftarrow}L\langle u\rangle]\rangle$

Proof. By induction on L. The base case is trivial. For $L = L'\langle\cdot\rangle[y{\leftarrow}w]$, by *i.h.* we have:

$$L'\langle t[x{\leftarrow}u]\rangle[y{\leftarrow}w] \equiv L'\langle t[x{\leftarrow}L'\langle u\rangle]\rangle[y{\leftarrow}w]$$

Let $(L'\langle u\rangle)_{[z]_y}$ be the result of replacing *all* occurrences of y by z in $L'\langle u\rangle$. Then:

$$\begin{array}{ll}
 & L'\langle t[x{\leftarrow}L'\langle u\rangle]\rangle[y{\leftarrow}w] \\
\equiv_{dup} & L'\langle t[x{\leftarrow}(L'\langle u\rangle)_{[z]_y}]\rangle[y{\leftarrow}w][z{\leftarrow}w] \\
\equiv^*_{com} & L'\langle t[x{\leftarrow}(L'\langle u\rangle)_{[z]_y}][z{\leftarrow}w]\rangle[y{\leftarrow}w] \\
\equiv_{[\cdot]} & L'\langle t[x{\leftarrow}(L'\langle u\rangle)_{[z]_y}[z{\leftarrow}w]]\rangle[y{\leftarrow}w] \\
=_\alpha & L'\langle t[x{\leftarrow}L'\langle u\rangle[y{\leftarrow}w]]\rangle[y{\leftarrow}w]
\end{array}$$

□ □

Now we prove the strong bisimulation property, by induction on $\multimap$.

1. **Base case 1: multiplicative root step** $t = L\langle\lambda x.t'\rangle L'\langle v\rangle \mapsto_{\mathtt{dBv}} u = L\langle t'[x{\leftarrow}L'\langle v\rangle]\rangle$**.** The nontrivial cases are when the $\Leftarrow\Rightarrow$ step overlaps the pattern of the $\mathtt{dBv}$-redex. Note that by Lemma B.3.1, if the $\Leftrightarrow$ is internal to $L'\langle v\rangle$, the proof is direct, since the $\mathtt{dBv}$-redex is preserved. More precisely, if $L'\langle v\rangle \Leftrightarrow L''\langle v'\rangle$, we have:

$$\begin{tikzcd}
L\langle\lambda x.t'\rangle\, L'\langle v\rangle \arrow[r, "\mathtt{dBv}"] \arrow[d, phantom, "\Leftrightarrow"] & L\langle t'[x{\leftarrow}L'\langle v\rangle]\rangle \arrow[d, phantom, "\Leftrightarrow"] \\
L\langle\lambda x.t'\rangle\, L''\langle v'\rangle \arrow[r, dashed, "\mathtt{dBv}"] & L\langle t'[x{\leftarrow}L''\langle v'\rangle]\rangle
\end{tikzcd}$$

Consider the remaining possibilities for $\Leftrightarrow$:

(a) *Garbage collection* $\equiv_{gc}$. The garbage collected substitution must be in L, *i.e.* L must be of the form $L_1\langle L_2[y{\leftarrow}L''\langle v'\rangle]\rangle$ with $y \notin \mathtt{fv}(L_2\langle\lambda x.t'\rangle)$. Let $\widehat{L} := L_1\langle L_2\rangle$. Then:

$$\begin{tikzcd}
L\langle\lambda x.t'\rangle\, L'\langle v\rangle \arrow[r, "\mathtt{dBv}"] \arrow[d, phantom, "\equiv_{gc}"] & L\langle t'[x{\leftarrow}L'\langle v\rangle]\rangle \arrow[d, phantom, "\equiv_{gc}"] \\
\widehat{L}\langle\lambda x.t'\rangle\, L'\langle v\rangle \arrow[r, dashed, "\mathtt{dBv}"] & \widehat{L}\langle t'[x{\leftarrow}L'\langle v\rangle]\rangle
\end{tikzcd}$$

(b) *Duplication* $\equiv_{dup}$. The duplicated substitution must be in L, *i.e.* L must be of the form $L_1\langle L_2[y{\leftarrow}u']\rangle$. Let $\widehat{L} := L_1\langle\langle\cdot\rangle[y{\leftarrow}u'][z{\leftarrow}u']\rangle$. Then:

$$\begin{tikzcd}
L\langle\lambda x.t'\rangle\, L'\langle v\rangle \arrow[r, "\mathtt{dBv}"] \arrow[d, phantom, "\equiv_{dup}"] & L\langle t'[x{\leftarrow}L'\langle v\rangle]\rangle \arrow[d, phantom, "\equiv_{dup}"] \\
\widehat{L}\langle(L_2\langle\lambda x.t'\rangle)_{[z]_y}\rangle\, L'\langle v\rangle \arrow[r, dashed, "\mathtt{dBv}"] & t_1
\end{tikzcd}$$

where $t_1 := \widehat{L}\langle(L_2\langle t'[x{\leftarrow}L'\langle v\rangle]\rangle)_{[z]_y}\rangle$.

(c) *Commutation with application* $\equiv_@$. The axiom can be applied only in one direction and there must be the same explicit substitution $[y{\leftarrow}q]$ as topmost constructor of each of the two sides of the application. The diagram is:

$$\begin{tikzcd}
L\langle\lambda x.t'\rangle[x{\leftarrow}q]\, L'\langle v\rangle[y{\leftarrow}q] \arrow[r, "\mathtt{dBv}"] \arrow[d, phantom, "\equiv_@"] & t_1 \arrow[d, phantom, "\equiv"] \\
(L\langle\lambda x.t'\rangle\, L'\langle v\rangle)[y{\leftarrow}q] \arrow[r, dashed, "\mathtt{dBv}"] & t_2
\end{tikzcd}$$

where

$$\begin{array}{l}
t_1 := L\langle t'[x{\leftarrow}L'\langle v\rangle[y{\leftarrow}q]]\rangle[x{\leftarrow}q], \\
t_2 := L\langle t'[x{\leftarrow}L'\langle v\rangle]\rangle[y{\leftarrow}q].
\end{array}$$

To prove the equivalence on the right, let $L'\langle v\rangle_{[z]_x}$ denote the result of replacing *all* occurrences of x by a fresh variable z in $L'\langle v\rangle$. The equivalence holds because:

$$\begin{array}{ll}
 & L\langle t''[y{\leftarrow}L'\langle v\rangle]\rangle[x{\leftarrow}q] \\
\equiv_{dup} & L\langle t''[y{\leftarrow}L'\langle v\rangle_{[z]_x}]\rangle[x{\leftarrow}q][z{\leftarrow}q] \\
\equiv^*_{com} & L\langle t''[y{\leftarrow}L'\langle v\rangle_{[z]_x}][z{\leftarrow}q]\rangle[x{\leftarrow}q] \\
\equiv_{[\cdot]} & L\langle t''[y{\leftarrow}L'\langle v\rangle_{[z]_x}[z{\leftarrow}q]]\rangle[x{\leftarrow}q] \\
=_\alpha & L\langle t''[y{\leftarrow}L'\langle v\rangle[x{\leftarrow}q]]\rangle[x{\leftarrow}q]
\end{array}$$

(d) *Commutation of independent substitutions* $\equiv_{com}$. The commutation of substitutions must be in L, *i.e.* L must be of the form $L_1\langle L_2[y{\leftarrow}u'][z{\leftarrow}w']\rangle$ with $z \notin \mathtt{fv}(u')$. Let $\widehat{L} := L_1\langle L_2[z{\leftarrow}w'][y{\leftarrow}u']\rangle$. Then:

$$\begin{tikzcd}
L\langle\lambda x.t'\rangle\, L'\langle v\rangle \arrow[r, "\mathtt{dBv}"] \arrow[d, phantom, "\equiv_{com}"] & L\langle t'[x{\leftarrow}L'\langle v\rangle]\rangle \arrow[d, phantom, "\equiv_{com}"] \\
\widehat{L}\langle\lambda x.t'\rangle\, L'\langle v\rangle \arrow[r, dashed, "\mathtt{dBv}"] & \widehat{L}\langle t'[x{\leftarrow}L'\langle v\rangle]\rangle
\end{tikzcd}$$

(e) *Composition of substitutions* $\equiv_{[\cdot]}$. The composition of substitutions must be in L, *i.e.* L must be of the form $L_1\langle L_2[y{\leftarrow}u'][z{\leftarrow}w']\rangle$ with $z \notin \mathtt{fv}(L_2\langle\lambda x.t'\rangle)$. Let $\widehat{L} := L_1\langle L_2[y{\leftarrow}u'[z{\leftarrow}w']]\rangle$. As in the previous case:

$$\begin{tikzcd}
L\langle\lambda x.t'\rangle\, L'\langle v\rangle \arrow[r, "\mathtt{dBv}"] \arrow[d, phantom, "\equiv_{[\cdot]}"] & L\langle t'[x{\leftarrow}L'\langle v\rangle]\rangle \arrow[d, phantom, "\equiv_{[\cdot]}"] \\
\widehat{L}\langle\lambda x.t'\rangle\, L'\langle v\rangle \arrow[r, dashed, "\mathtt{dBv}"] & \widehat{L}\langle t'[x{\leftarrow}L'\langle v\rangle]\rangle
\end{tikzcd}$$

2. **Base case 2: exponential root step** $t = V\langle x\rangle[x{\leftarrow}L\langle v\rangle] \mapsto_{\mathtt{lsv}} u = L\langle V\langle v\rangle[x{\leftarrow}v]\rangle$**.** Consider first the case when the $\Leftrightarrow$-redex is internal to $V\langle x\rangle$. By Lemma B.3.2 we know $\Leftrightarrow$ preserves the shape of $V\langle x\rangle$, *i.e.* $V\langle x\rangle \Leftrightarrow \widehat{V}\langle x\rangle$. Then:

$$\begin{tikzcd}
V\langle x\rangle[x{\leftarrow}L\langle v\rangle] \arrow[r, "\mathtt{lsv}"] \arrow[d, phantom, "\Leftrightarrow"] & L\langle V\langle v\rangle[x{\leftarrow}v]\rangle \arrow[d, phantom, "\equiv"] \\
\widehat{V}\langle x\rangle[x{\leftarrow}L\langle v\rangle] \arrow[r, dashed, "\mathtt{lsv}"] & L\langle\widehat{V}\langle v\rangle[x{\leftarrow}v]\rangle
\end{tikzcd}$$

If the $\Leftrightarrow$-redex is internal to one of the substitutions in L, the proof is straightforward. Note that the $\Leftrightarrow$-redex has always a substitution at the root. The remaining possibilities are such that substitution is in L, or that it is precisely $[x{\leftarrow}L\langle v\rangle]$. Axiom by axiom:

(a) *Garbage collection* $\equiv_{gc}$. If the garbage collected substitution is in L, let $\widehat{L}$ be L without such substitution. Then:

$$\begin{tikzcd}
V\langle x\rangle[x{\leftarrow}L\langle v\rangle] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{gc}"] & L\langle V\langle v\rangle[x{\leftarrow}v]\rangle \arrow[d, phantom, "\equiv_{gc}"] \\
V\langle x\rangle[x{\leftarrow}\widehat{L}\langle v\rangle] \arrow[r, multimap, dashed, "\mathtt{lsv}"] & \widehat{L}\langle V\langle v\rangle[x{\leftarrow}v]\rangle
\end{tikzcd}$$

The garbage collected substitution cannot be $[x{\leftarrow}L\langle v\rangle]$, since this would imply $x \notin \mathtt{fv}(V\langle x\rangle)$, which is a contradiction.

(b) *Duplication* $\equiv_{dup}$. If the duplicated substitution is in L, then L is of the form $L_1\langle L_2[y{\leftarrow}t']\rangle$. Let $\widehat{L} = L_1\langle [y{\leftarrow}t'][z{\leftarrow}t']\rangle$. Then:

$$\begin{tikzcd}
V\langle x\rangle[x{\leftarrow}L\langle v\rangle] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{dup}"] & L\langle V\langle v\rangle[x{\leftarrow}v]\rangle \arrow[d, phantom, "\equiv_{dup}"] \\
t_1 \arrow[r, multimap, dashed, "\mathtt{lsv}"] & t_2
\end{tikzcd}$$

where

$$\begin{aligned}
t_1 &:= V\langle x\rangle[x{\leftarrow}\widehat{L}\langle L_{2[z]_y}\langle v_{[z]_y}\rangle\rangle],\\
t_2 &:= \widehat{L}\langle L_{2[z]_y}\langle V\langle v_{[z]_y}\rangle[x{\leftarrow}v_{[z]_y}]\rangle\rangle.
\end{aligned}$$

If the duplicated substitution is $[x{\leftarrow}L\langle v\rangle]$, there are two possibilities, depending on whether the occurrence of x substituted by the $\mapsto_{\mathtt{lsv}}$ step is replaced by the fresh variable y, or left untouched. If it is not replaced:

$$\begin{tikzcd}
V\langle x\rangle[x{\leftarrow}L\langle v\rangle] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[dd, phantom, "\equiv_{dup}"] & L\langle V\langle v\rangle[x{\leftarrow}v]\rangle \arrow[d, phantom, "\equiv_{dup}"] \\
 & t_2 \arrow[d, phantom, "\equiv\ \text{(Lemma B.4)}"] \\
t_4 \arrow[r, multimap, dashed, "\mathtt{lsv}"] & t_3
\end{tikzcd}$$

where

$$\begin{aligned}
t_2 &:= L\langle (V\langle v\rangle)_{[y]_x}[x{\leftarrow}v][y{\leftarrow}v]\rangle,\\
t_3 &:= L\langle (V\langle v\rangle)_{[y]_x}[x{\leftarrow}v][y{\leftarrow}L\langle v\rangle]\rangle,\\
t_4 &:= (V\langle x\rangle)_{[y]_x}[x{\leftarrow}L\langle v\rangle][y{\leftarrow}L\langle v\rangle].
\end{aligned}$$

If the occurrence of x substituted by the $\mapsto_{\mathtt{lsv}}$ step is replaced by the fresh variable y, the situation is essentially analogous.

(c) *Commutation with application* $\equiv_{@}$. The only possibility is that the substitution $[x{\leftarrow}L\langle v\rangle]$ is commuted with the outermost application in $V\langle x\rangle$. Two cases:

i. *The substitution acts on the left of the application, i.e.* $V = V' t'$.

$$\begin{tikzcd}
(V'\langle x\rangle\, t')[x{\leftarrow}L\langle v\rangle] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[dd, phantom, "\equiv_{@}"] & t_1 \arrow[d, phantom, "\equiv_{@}^{*}"] \\
 & t_2 \arrow[d, phantom, "\equiv_{[\cdot]}^{*}"] \\
t_3 \arrow[r, multimap, dashed, "\mathtt{lsv}"] & t_4
\end{tikzcd}$$

where

$$\begin{aligned}
t_1 &:= L\langle (V'\langle v\rangle\, t')[x{\leftarrow}v]\rangle,\\
t_2 &:= L\langle V'\langle v\rangle[x{\leftarrow}v]\rangle L\langle t'[x{\leftarrow}v]\rangle,\\
t_3 &:= V'\langle x\rangle[x{\leftarrow}L\langle v\rangle]t'[x{\leftarrow}L\langle v\rangle],\\
t_4 &:= L\langle V'\langle v\rangle[x{\leftarrow}v]\rangle t'[x{\leftarrow}L\langle v\rangle].
\end{aligned}$$

ii. *The substitution acts on the right of the application, i.e.* $V = L'\langle v'\rangle V'$. Similar to the previous case:

$$\begin{tikzcd}
(L'\langle v'\rangle\, V'\langle x\rangle)[x{\leftarrow}L\langle v\rangle] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[dd, phantom, "\equiv_{@}"] & t_1 \arrow[d, phantom, "\equiv_{@}^{*}"] \\
 & t_2 \arrow[d, phantom, "\equiv_{[\cdot]}^{*}"] \\
t_3 \arrow[r, multimap, dashed, "\mathtt{lsv}"] & t_4
\end{tikzcd}$$

where

$$\begin{aligned}
t_1 &:= L\langle (L'\langle v'\rangle\, V'\langle v\rangle)[x{\leftarrow}v]\rangle,\\
t_2 &:= L\langle L'\langle v'\rangle[x{\leftarrow}v]\rangle L\langle V'\langle v\rangle[x{\leftarrow}v]\rangle,\\
t_3 &:= L'\langle v'\rangle[x{\leftarrow}L\langle v\rangle]V'\langle x\rangle[x{\leftarrow}L\langle v\rangle],\\
t_4 &:= L'\langle v'\rangle[x{\leftarrow}L\langle v\rangle]L\langle V'\langle v\rangle[x{\leftarrow}v]\rangle.
\end{aligned}$$

(d) *Commutation of independent substitutions* $\equiv_{com}$. If the commuted substitutions both belong to L, let $\widehat{L}$ be the result of commuting them, and the situation is exactly as for Garbage collection.
The remaining possibility is that $V = V'[y{\leftarrow}t']$ and $[x{\leftarrow}L\langle v\rangle]$ commutes with $[y{\leftarrow}t']$ (which implies $x \notin \mathtt{fv}(t')$). Then:

$$\begin{tikzcd}
V'\langle x\rangle[y{\leftarrow}t'][x{\leftarrow}L\langle v\rangle] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{com}"] & L\langle V'\langle v\rangle[y{\leftarrow}t'][x{\leftarrow}v]\rangle \arrow[d, phantom, "\equiv_{com}^{*}"] \\
V'\langle x\rangle[x{\leftarrow}L\langle v\rangle][y{\leftarrow}t'] \arrow[r, multimap, dashed, "\mathtt{lsv}"] & L\langle V'\langle v\rangle[x{\leftarrow}v]\rangle[y{\leftarrow}t']
\end{tikzcd}$$

(e) *Composition of substitutions* $\equiv_{[\cdot]}$. If the composed substitutions both belong to L, let $\widehat{L}$ be the result of composing them, and the situation is exactly as for Garbage collection.
The remaining possibility is that $[x{\leftarrow}L\langle v\rangle]$ is the outermost substitution composed by $\equiv_{[\cdot]}$. This is not possible if the rule is applied from left to right, since it would imply that $V\langle x\rangle = V'\langle x\rangle[y{\leftarrow}t']$ with $x \notin V'\langle x\rangle$, which is a contradiction.
Finally, if the $\equiv_{[\cdot]}$ rule is applied from right to left, L is of the form $L'[y{\leftarrow}t']$ and:

$$\begin{tikzcd}
V\langle x\rangle[x{\leftarrow}L'\langle v\rangle[y{\leftarrow}t']] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{[\cdot]}"] & L'\langle V\langle v\rangle[x{\leftarrow}v]\rangle[y{\leftarrow}t'] \arrow[d, phantom, "="] \\
V\langle x\rangle[x{\leftarrow}L'\langle v\rangle][y{\leftarrow}t'] \arrow[r, multimap, dashed, "\mathtt{lsv}"] & L'\langle V\langle x\rangle[x{\leftarrow}v]\rangle[y{\leftarrow}t']
\end{tikzcd}$$

3. **Inductive case 1: left of an application** $V = V'q$**.** The situation is:

$$t = V'\langle t'\rangle\, q \multimap V'\langle u'\rangle\, q = u$$

If the $\Longleftrightarrow$ step is internal to $V'\langle t'\rangle$, the result follows by *i.h.*. The proof is also direct if $\Longleftrightarrow$ is internal to q. The nontrivial case is when the $\Longleftrightarrow$ step overlaps $V'\langle t'\rangle$ and q. There are two possibilities. The first is trivial: $\equiv_{gc}$ is used to introduce a substitution out of the blue, but this case clearly commutes with reduction.

The second is that the application is commuted with a substitution via the $\equiv_{@}$ rule (applied from right to left). There are two cases:

(a) *The substitution comes from* t'. That is, $V' = \langle\cdot\rangle$ and t' has a substitution at its root. Then t' must be a $\mapsto_{\mathtt{lsv}}$-redex $t' = V''\langle x\rangle[x{\leftarrow}L\langle v\rangle]$. Moreover $q = q'[x{\leftarrow}L\langle v\rangle]$. We have:

$$\begin{array}{ccc} V''\langle x\rangle[x{\leftarrow}L\langle v\rangle]\, q'[x{\leftarrow}L\langle v\rangle] & \xrightarrow{\mathtt{lsv}}\!\!\circ & t_1 \\ \equiv_{@} & & \equiv \\ t_2 & \overset{\mathtt{lsv}}{\dashrightarrow}\!\!\circ & t_3 \end{array}$$

where

$$\begin{aligned} t_1 &:= L\langle V''\langle v\rangle[x{\leftarrow}v]\rangle\, q'[x{\leftarrow}L\langle v\rangle], \\ t_2 &:= (V''\langle x\rangle\, q')[x{\leftarrow}L\langle v\rangle], \\ t_3 &:= L\langle (V''\langle v\rangle\, q')[x{\leftarrow}v]\rangle. \end{aligned}$$

For the equivalence on the right note that:

$$\begin{array}{ll} & L\langle V''\langle v\rangle[x{\leftarrow}v]\rangle\, q'[x{\leftarrow}L\langle v\rangle] \\ \equiv^*_{[\cdot]} & L\langle V''\langle v\rangle[x{\leftarrow}v]\rangle\, L\langle q'[x{\leftarrow}v]\rangle \\ \equiv^*_{@} & L\langle V''\langle v\rangle[x{\leftarrow}v]\, q'[x{\leftarrow}v]\rangle \\ \equiv_{@} & L\langle (V''\langle v\rangle\, q')[x{\leftarrow}v]\rangle \end{array}$$

(b) *The substitution comes from* V'. That is: $V' = V''[x{\leftarrow}w']$. Moreover, $q = q'[x{\leftarrow}w']$. The proof is then straightforward:

$$\begin{array}{ccc} V''\langle t'\rangle[x{\leftarrow}w']\, q'[x{\leftarrow}w'] & \longrightarrow\!\!\circ & t_1 \\ \equiv_{@} & & \equiv_{@} \\ t_2 & \dashrightarrow\!\!\circ & t_3 \end{array}$$

where

$$\begin{aligned} t_1 &:= V''\langle u'\rangle[x{\leftarrow}w']\, q'[x{\leftarrow}w'], \\ t_2 &:= (V''\langle t'\rangle\, q')[x{\leftarrow}w'], \\ t_3 &:= (V''\langle u'\rangle\, q')[x{\leftarrow}w']. \end{aligned}$$

4. **Inductive case 2: right of an application** $V = L\langle v\rangle V'$**.** The situation is:

$$t = L\langle v\rangle\, V'\langle t'\rangle \multimap L\langle v\rangle\, V'\langle u'\rangle = u$$

Reasoning as in the previous case (*left of an application*), if the $\Longleftrightarrow$ step is internal to $V'\langle t'\rangle$, the result follows by *i.h.*, and if it is internal to $L\langle v\rangle$, it is straightforward to close the diagram by resorting to the fact that $\equiv$ preserves the shape of $L\langle v\rangle$ (Lemma B.3).
The remaining possibility is that the $\Longleftrightarrow$ step overlaps both $L\langle v\rangle$ and $V'\langle t'\rangle$. As in the previous case, this can only be possible if $\equiv_{gc}$ introduces a substitution out of the blue, which is a trivial case, or because of a *Commutation with application* rule ($\equiv_{@}$, from right to left). This again leaves two possibilities:

(a) *The substitution comes from* t'. That is, $V' = \langle\cdot\rangle$ and t' is a $\mapsto_{\mathtt{lsv}}$-redex $t' = V''\langle y\rangle[y{\leftarrow}L'\langle v'\rangle]$. Moreover, $L = L''[y{\leftarrow}L'\langle v'\rangle]$. Then:

$$\begin{array}{ccc} L''\langle v\rangle[y{\leftarrow}L'\langle v'\rangle]\, V''\langle y\rangle[y{\leftarrow}L'\langle v'\rangle] & \xrightarrow{\mathtt{lsv}}\!\!\circ & t_1 \\ \equiv_{@} & & \equiv \\ t_2 & \overset{\mathtt{lsv}}{\dashrightarrow}\!\!\circ & t_3 \end{array}$$

where

$$\begin{aligned} t_1 &:= L''\langle v\rangle[y{\leftarrow}L'\langle v'\rangle]\, L'\langle V''\langle v'\rangle[y{\leftarrow}v']\rangle, \\ t_2 &:= (L''\langle v\rangle\, V''\langle y\rangle)[y{\leftarrow}L'\langle v'\rangle], \\ t_3 &:= L'\langle (L''\langle v\rangle\, V''\langle v'\rangle)[y{\leftarrow}v']\rangle. \end{aligned}$$

Exactly as in the previous case, for the equivalence on the right consider:

$$\begin{array}{ll} & L''\langle v\rangle[y{\leftarrow}L'\langle v'\rangle]\, L'\langle V''\langle v'\rangle[y{\leftarrow}v']\rangle \\ \equiv^*_{[\cdot]} & L'\langle L''\langle v\rangle[y{\leftarrow}v']\rangle\, L'\langle V''\langle v'\rangle[y{\leftarrow}v']\rangle \\ \equiv^*_{@} & L'\langle L''\langle v\rangle[y{\leftarrow}v']\, V''\langle v'\rangle[y{\leftarrow}v']\rangle \\ \equiv_{@} & L'\langle (L''\langle v\rangle\, V''\langle v'\rangle)[y{\leftarrow}v']\rangle \end{array}$$

(b) *The substitution comes from* V'. That is, $V' = V''[x{\leftarrow}w']$. Moreover, $L = L'[x{\leftarrow}w']$. This case is then straightforward:

$$\begin{array}{ccc} L'\langle v\rangle[x{\leftarrow}w']\, V''\langle t'\rangle[x{\leftarrow}w'] & \longrightarrow\!\!\circ & L'\langle v\rangle[x{\leftarrow}w']\, V''\langle u'\rangle[x{\leftarrow}w'] \\ \equiv_{@} & & \equiv_{@} \\ (L'\langle v\rangle\, V''\langle t'\rangle)[x{\leftarrow}w'] & \dashrightarrow\!\!\circ & (L'\langle v\rangle\, V''\langle u'\rangle)[x{\leftarrow}w'] \end{array}$$

5. **Inductive case 3: left of a substitution** $V = V'[x{\leftarrow}q]$**.** The situation is:

$$t = V'\langle t'\rangle[x{\leftarrow}q] \multimap V'\langle u'\rangle[x{\leftarrow}q] = u$$

If the $\Longleftrightarrow$ step is internal to $V'\langle t'\rangle$, the result follows by *i.h.*. If it is internal to q, the steps are orthogonal, which makes the diagram trivial. If the equivalence $\equiv_{gc}$ introduces a substitution out of the blue the steps trivially commute.
The remaining possibility is that the substitution $[x{\leftarrow}q]$ is involved in the $\Longleftrightarrow$ redex. By case analysis on the kind of the step $\equiv_b$:

(a) *Garbage collection* $\equiv_{gc}$. We know $x \notin \mathtt{fv}(V'\langle t'\rangle)$ and therefore also $x \notin \mathtt{fv}(V'\langle u'\rangle)$. We get:

$$\begin{array}{ccc} V'\langle t'\rangle[x{\leftarrow}q] & \longrightarrow\!\!\circ & V'\langle u'\rangle[x{\leftarrow}q] \\ \equiv_{gc} & & \equiv_{gc} \\ V'\langle t'\rangle & \dashrightarrow\!\!\circ & V'\langle u'\rangle \end{array}$$

(b) *Duplication* $\equiv_{dup}$. The important fact is that if $V'\langle t'\rangle \multimap V'\langle u'\rangle$ and $V'\langle t'\rangle_{[y]_x}$ denotes the result of renaming some (arbitrary) occurrences of x by y in $V'\langle t'\rangle$, then $V'\langle t'\rangle_{[y]_x} \multimap V'\langle u'\rangle_{[y]_x}$, where $V'\langle u'\rangle_{[y]_x}$ denotes the result of renaming some occurrences of x by y in $V'\langle u'\rangle$. By this we conclude:

$$\begin{array}{ccc} V'\langle t'\rangle[x{\leftarrow}q] & \longrightarrow\!\!\circ & V'\langle u'\rangle[x{\leftarrow}q] \\ \equiv_{dup} & & \equiv_{dup} \\ (V'\langle t'\rangle)_{[y]_x}[x{\leftarrow}q][y{\leftarrow}q] & \dashrightarrow\!\!\circ & (V'\langle u'\rangle)_{[y]_x}[x{\leftarrow}q][y{\leftarrow}q] \end{array}$$

(c) *Commutation with application* $\equiv_{@}$. $V'\langle t'\rangle$ must be an application. This allows for three possibilities:

i. *The application comes from* t'. That is, $V' = \langle\cdot\rangle$ and t' is a $\mapsto_{\mathtt{dBv}}$-redex $t' = L\langle\lambda y.t''\rangle\, L'\langle v\rangle$. The diagram is exactly as for the multiplicative base case 1c (read bottom-up).

ii. *The application comes from V', left case.* That is, $V' = V''\,w'$. This case is direct:

$$\begin{tikzcd}
(V''\langle t'\rangle\, w')[x\leftarrow q] \arrow[r, multimap] \arrow[d, phantom, "\equiv_{@}"] & t_1 \arrow[d, phantom, "\equiv_{@}"] \\
t_2 \arrow[r, dashed, multimap] & t_3
\end{tikzcd}$$

where

$$\begin{aligned}
t_1 &:= (V''\langle u'\rangle\, w')[x\leftarrow q],\\
t_2 &:= V''\langle t'\rangle[x\leftarrow q]\, w'[x\leftarrow q],\\
t_3 &:= V''\langle u'\rangle[x\leftarrow q]\, w'[x\leftarrow q].
\end{aligned}$$

iii. *The application comes from V', right case.* That is, $V' = L\langle v\rangle\, V''$. Analogous to the previous case.

(d) *Commutation of independent substitutions $\equiv_{com}$.* Since $V'\langle t'\rangle$ must have a substitution at the root, there are two possibilities:

i. *The substitution comes from t'.* That is, $V' = \langle\cdot\rangle$ and t' is a $\mapsto_{\mathtt{lsv}}$-redex $t' = V''\langle y\rangle[y\leftarrow L\langle v\rangle]$, with $x \notin \mathtt{fv}(L\langle v\rangle)$. Then:

$$\begin{tikzcd}
V''\langle y\rangle[y\leftarrow L\langle v\rangle][x\leftarrow q] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{com}"] & L\langle V''\langle v\rangle[y\leftarrow v]\rangle[x\leftarrow q] \arrow[d, phantom, "\equiv_{com}^{*}"] \\
V''\langle y\rangle[x\leftarrow q][y\leftarrow L\langle v\rangle] \arrow[r, dashed, multimap, "\mathtt{lsv}"] & L\langle V''\langle v\rangle[x\leftarrow q][y\leftarrow v]\rangle
\end{tikzcd}$$

ii. *The substitution comes from V'.* That is, $V' = V''[y\leftarrow w']$ with $x \notin \mathtt{fv}(w')$. This case is direct:

$$\begin{tikzcd}
V''\langle t'\rangle[y\leftarrow w'][x\leftarrow q] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{com}"] & V''\langle u'\rangle[y\leftarrow w'][x\leftarrow q] \arrow[d, phantom, "\equiv_{com}"] \\
V''\langle t'\rangle[x\leftarrow q][y\leftarrow w'] \arrow[r, dashed, multimap, "\mathtt{lsv}"] & V''\langle u'\rangle[x\leftarrow q][y\leftarrow w']
\end{tikzcd}$$

(e) *Composition of substitutions $\equiv_{[\cdot]}$.* As in the previous case, there are two possibilities:

i. *The substitution comes from t'.* That is, $V' = \langle\cdot\rangle$ and t' is a $\mapsto_{\mathtt{lsv}}$-redex $t' = V''\langle y\rangle[y\leftarrow L\langle v\rangle]$, with $x \notin \mathtt{fv}(V''\langle y\rangle)$. Then:

$$\begin{tikzcd}
V''\langle y\rangle[y\leftarrow L\langle v\rangle][x\leftarrow q] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{[\cdot]}"] & L\langle V''\langle v\rangle[y\leftarrow v]\rangle[x\leftarrow q] \arrow[d, phantom, "="] \\
V''\langle y\rangle[y\leftarrow L\langle v\rangle[x\leftarrow q]] \arrow[r, dashed, multimap, "\mathtt{lsv}"] & L\langle V''\langle v\rangle[y\leftarrow v]\rangle[x\leftarrow q]
\end{tikzcd}$$

ii. *The substitution comes from V'.* That is, $V' = V''[y\leftarrow w']$ with $x \notin \mathtt{fv}(V''\langle t'\rangle)$. The proof for this case is direct:

$$\begin{tikzcd}
V''\langle t'\rangle[y\leftarrow w'][x\leftarrow q] \arrow[r, multimap] \arrow[d, phantom, "\equiv_{[\cdot]}"] & V''\langle u'\rangle[y\leftarrow w'][x\leftarrow q] \arrow[d, phantom, "\equiv_{[\cdot]}"] \\
V''\langle t'\rangle[y\leftarrow w'[x\leftarrow q]] \arrow[r, dashed, multimap] & V''\langle u'\rangle[y\leftarrow w'[x\leftarrow q]]
\end{tikzcd}$$

B.3 Proof of Proposition 2.2 ($\equiv$ is a strong bisimulation) for right-to-left call-by-value

The proof is obtained as a minimal variation over the proof for left-to-right call-by-value (previous subsection), and is therefore omitted.

B.4 Proof of Proposition 2.2 ($\equiv$ is a strong bisimulation) for call-by-need

We need two preliminary lemmas, proved by straightforward inductions on N:

Lemma B.5. *Let t be a term, N be a call-by-need evaluation context not capturing any variable in $\mathtt{fv}(t)$, and $x \notin \mathtt{fv}(N\langle y\rangle)$. Then $N\langle t[x\leftarrow u]\rangle \equiv N\langle t\rangle[x\leftarrow u]$.*

Lemma B.6. *The equivalence relation $\equiv$ as defined for call-by-need preserves the shape of $N\langle x\rangle$. More precisely, if $N\langle x\rangle \equiv t$, with x not captured by N, then t is of the form $N'\langle x\rangle$, with x not captured by N'.*

We follow the structure of the previous proofs of strong bisimulation, in particular the proof is by induction on $\multimap$. Remember that for call-by-need the definition of the structural equivalence is different, it is the one given only by axioms $\equiv_{@l}$, $\equiv_{com}$, and $\equiv_{[\cdot]}$.

1. **Base case 1: multiplicative root step** $t = L\langle \lambda x.t'\rangle q \mapsto_{\mathtt{dB}} u = L\langle t'[x\leftarrow q]\rangle$**.** Every application of $\equiv$ inside q or inside one of the substitutions in L trivially commutes with the step. The interesting cases are those where structural equivalence has a critical pair with the step:

(a) *Commutation with left of an application $\equiv_{@l}$.* If $L = L'[y\leftarrow r]$ then

$$\begin{tikzcd}
L'\langle \lambda x.t'\rangle[y\leftarrow r]q \arrow[r, multimap, "\mathtt{dB}"] \arrow[d, phantom, "\equiv_{@l}"] & L'\langle t'[x\leftarrow q]\rangle[y\leftarrow r] \arrow[d, phantom, "="] \\
(L'\langle \lambda x.t'\rangle q)[y\leftarrow r] \arrow[r, dashed, multimap, "\mathtt{dB}"] & L'\langle t'[x\leftarrow q]\rangle[y\leftarrow r]
\end{tikzcd}$$

(b) *Commutation of independent substitutions $\equiv_{com}$.* The substitutions that are commuted by the $\equiv_{com}$ rule must be both in L, *i.e.* L must be of the form $L'\langle L''[y\leftarrow w'][z\leftarrow r']\rangle$ with $z \notin \mathtt{fv}(w')$. Let $\widehat{L} = L'\langle L''[z\leftarrow r'][y\leftarrow w']\rangle$. Then:

$$\begin{tikzcd}
L\langle \lambda x.t'\rangle u' \arrow[r, multimap, "\mathtt{dB}"] \arrow[d, phantom, "\equiv_{com}"] & L\langle t'[x\leftarrow u']\rangle \arrow[d, phantom, "\equiv_{com}"] \\
\widehat{L}\langle \lambda x.t'\rangle u' \arrow[r, dashed, multimap, "\mathtt{dB}"] & \widehat{L}\langle t'[x\leftarrow u']\rangle
\end{tikzcd}$$

(c) *Composition of substitutions $\equiv_{[\cdot]}$.* The substitutions that appear in the left-hand side of the $\equiv_{[\cdot]}$ rule must both be in L, *i.e.* L must be of the form $L'\langle L''[y\leftarrow w'][z\leftarrow r']\rangle$ with $z \notin \mathtt{fv}(L''\langle \lambda x.t'\rangle)$. Let $\widehat{L} = L'\langle L''[y\leftarrow w'[z\leftarrow r']]\rangle$. Exactly as in the previous case:

$$\begin{tikzcd}
L\langle \lambda x.t'\rangle u' \arrow[r, multimap, "\mathtt{dB}"] \arrow[d, phantom, "\equiv_{[\cdot]}"] & L\langle t'[x\leftarrow u']\rangle \arrow[d, phantom, "\equiv_{[\cdot]}"] \\
\widehat{L}\langle \lambda x.t'\rangle u' \arrow[r, dashed, multimap, "\mathtt{dB}"] & \widehat{L}\langle t'[x\leftarrow u']\rangle
\end{tikzcd}$$

2. **Base case 2: exponential root step** $t = N\langle x\rangle[x\leftarrow L\langle v\rangle] \mapsto_{\mathtt{lsv}} u = L\langle N\langle v\rangle[x\leftarrow v]\rangle$**.** Consider first the case when the $\Longleftrightarrow$-redex is internal to $N\langle x\rangle$. By Lemma B.6 we know $\Longleftrightarrow$ preserves the shape of $N\langle x\rangle$, *i.e.* $N\langle x\rangle \Longleftrightarrow \widehat{N}\langle x\rangle$. Then:

$$\begin{tikzcd}
N\langle x\rangle[x\leftarrow L\langle v\rangle] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\Longleftrightarrow"] & L\langle N\langle v\rangle[x\leftarrow v]\rangle \arrow[d, phantom, "="] \\
\widehat{N}\langle x\rangle[x\leftarrow L\langle v\rangle] \arrow[r, dashed, multimap, "\mathtt{lsv}"] & L\langle \widehat{N}\langle v\rangle[x\leftarrow v]\rangle
\end{tikzcd}$$

If the $\Longleftrightarrow$-redex is internal to one of the substitutions in L, the proof is straightforward. Note that the $\Longleftrightarrow$-redex has always a substitution at the root. The remaining possibilities are that such substitution is in L, or that it is precisely $[x{\leftarrow}L\langle v\rangle]$. Axiom by axiom:

(a) *Commutation with the left of an application* $\equiv_{@l}$. The only possibility is that the substitution $[x{\leftarrow}L\langle v\rangle]$ is commuted with the outermost application in $N\langle x\rangle$, *i.e.* $N = N't'$. The diagram is:

$$\begin{tikzcd}
(N'\langle x\rangle\, t')[x{\leftarrow}L\langle v\rangle] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{@l}"] & L\langle (N'\langle v\rangle\, t')[x{\leftarrow}v]\rangle \arrow[d, phantom, "\equiv^*_{@l}"] \\
N'\langle x\rangle[x{\leftarrow}L\langle v\rangle]t' \arrow[r, multimap, dashed, "\mathtt{lsv}"] & L\langle N'\langle v\rangle[x{\leftarrow}v]\rangle\, t'
\end{tikzcd}$$

(b) *Commutation of independent substitutions* $\equiv_{com}$. Two sub-cases:
   i. *The commuted substitutions both belong to* L. Let $\widehat{L}$ be the result of commuting them, and the diagram is:

$$\begin{tikzcd}
N\langle x\rangle[x{\leftarrow}L\langle v\rangle] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{com}"] & L\langle N\langle v\rangle[x{\leftarrow}v]\rangle \arrow[d, phantom, "\equiv_{com}"] \\
N\langle x\rangle[x{\leftarrow}\widehat{L}\langle v\rangle] \arrow[r, multimap, dashed, "\mathtt{lsv}"] & \widehat{L}\langle N\langle v\rangle[x{\leftarrow}v]\rangle
\end{tikzcd}$$

   ii. *One of the commuted substitutions is* $[x{\leftarrow}L\langle v\rangle]$. Then $N = N'[y{\leftarrow}t']$ and $[x{\leftarrow}L\langle v\rangle]$ commutes with $[y{\leftarrow}t']$ (which implies $x \notin \mathtt{fv}(t')$). Then:

$$\begin{tikzcd}
N'\langle x\rangle[y{\leftarrow}t'][x{\leftarrow}L\langle v\rangle] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{com}"] & L\langle N'\langle v\rangle[y{\leftarrow}t'][x{\leftarrow}v]\rangle \arrow[d, phantom, "\equiv^*_{com}"] \\
N'\langle x\rangle[x{\leftarrow}L\langle v\rangle][y{\leftarrow}t'] \arrow[r, multimap, dashed, "\mathtt{lsv}"] & L\langle N'\langle v\rangle[x{\leftarrow}v]\rangle[y{\leftarrow}t']
\end{tikzcd}$$

(c) *Composition of substitutions* $\equiv_{[\cdot]}$. Two sub-cases:
   i. *The composed substitutions both belong to* L. Analogous to case 2(b)i.
   ii. *One of the composed subtitutions is* $[x{\leftarrow}L\langle v\rangle]$. This is not possible if the rule is applied from left to right, since it would imply that $N\langle x\rangle = N'\langle x\rangle[y{\leftarrow}t']$ with $x \notin N'\langle x\rangle$, which is a contradiction.
   Finally, if the $\equiv_{[\cdot]}$ rule is applied from right to left, L is of the form $L'[y{\leftarrow}t']$ and:

$$\begin{tikzcd}
N\langle x\rangle[x{\leftarrow}L'\langle v\rangle[y{\leftarrow}t']] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{[\cdot]}"] & L'\langle N\langle v\rangle[x{\leftarrow}v]\rangle[y{\leftarrow}t'] \arrow[d, phantom, "="] \\
N\langle x\rangle[x{\leftarrow}L'\langle v\rangle][y{\leftarrow}t'] \arrow[r, multimap, dashed, "\mathtt{lsv}"] & L'\langle N\langle x\rangle[x{\leftarrow}v]\rangle[y{\leftarrow}t']
\end{tikzcd}$$

3. **Inductive case 1: left of an application** $N = N'q$**.** The situation is:

$$t = N'\langle t'\rangle\, q \multimap N'\langle u'\rangle\, q = u$$

If the $\Longleftrightarrow$ step is internal to $N'\langle t'\rangle$, the result follows by *i.h.*. The proof is also direct if $\Longleftrightarrow$ is internal to q. The nontrivial cases are those where $\Longleftrightarrow$ overlaps $N'\langle t'\rangle$ and q. The only possible case is that a substitution commutes with the topmost application via $\equiv_{@l}$ (applied from right to left). There are two cases:

(a) *The substitution comes from* t'. That is, $N' = \langle\cdot\rangle$ and t' has a substitution at its root. Then t' must be a $\mapsto_{\mathtt{lsv}}$-redex $t' = N''\langle x\rangle[x{\leftarrow}L\langle v\rangle]$. We have:

$$\begin{tikzcd}
N''\langle x\rangle[x{\leftarrow}L\langle v\rangle]\, q \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{@l}"] & L\langle N''\langle v\rangle[x{\leftarrow}v]\rangle\, q \arrow[d, phantom, "\equiv^*_{@l}"] \\
(N''\langle x\rangle\, q)[x{\leftarrow}L\langle v\rangle] \arrow[r, multimap, dashed, "\mathtt{lsv}"] & L\langle (N''\langle v\rangle\, q)[x{\leftarrow}v]\rangle
\end{tikzcd}$$

(b) *The substitution comes from* N'. That is: $N' = N''[x{\leftarrow}w']$. The proof is then straightforward:

$$\begin{tikzcd}
N''\langle t'\rangle[x{\leftarrow}w']\, q \arrow[r, multimap] \arrow[d, phantom, "\equiv_{@l}"] & N''\langle u'\rangle[x{\leftarrow}w']\, q \arrow[d, phantom, "\equiv_{@l}"] \\
(N''\langle t'\rangle\, q)[x{\leftarrow}w'] \arrow[r, multimap, dashed] & (N''\langle u'\rangle\, q)[x{\leftarrow}w']
\end{tikzcd}$$

4. **Inductive case 2: left of a substitution** $N = N'[x{\leftarrow}q]$**.** The situation is:

$$t = N'\langle t'\rangle[x{\leftarrow}q] \multimap N'\langle u'\rangle[x{\leftarrow}q] = u$$

If the $\Longleftrightarrow$ step is internal to $N'\langle t'\rangle$, the result follows by *i.h.*. If it is internal to q, the steps are orthogonal, which makes the diagram trivial. The remaining possibility is that the substitution $[x{\leftarrow}q]$ is involved in the $\Longleftrightarrow$ redex. By case analysis on the kind of the step $\equiv_b$:

(a) *Commutation with the left of an application* $\equiv_{@l}$. $N'\langle t'\rangle$ must be an application. Two possibilities:
   i. *The application comes from* t'. That is, $N' = \langle\cdot\rangle$ and t' is a $\mapsto_{\mathtt{dB}}$-redex $t' = L\langle \lambda y.t''\rangle\, r$. This is exactly as the base case 1a (read bottom-up).
   ii. *The application comes from* N', *i.e.* $N' = N''\, w'$. This is exactly as the inductive case 3b (read bottom-up).

(b) *Commutation of independent substitutions* $\equiv_{com}$. Since $N'\langle t'\rangle$ must have a substitution at the root, there are two possibilities:
   i. *The substitution comes from* t'. That is, $N' = \langle\cdot\rangle$ and t' is a $\mapsto_{\mathtt{lsv}}$-redex $t' = N''\langle y\rangle[y{\leftarrow}L\langle v\rangle]$, with $x \notin \mathtt{fv}(L\langle v\rangle)$. This case is exactly as the base exponential case 2(b)ii (read bottom-up).
   ii. *The substitution comes from* N'. That is, $N' = N''[y{\leftarrow}w']$ with $x \notin \mathtt{fv}(w')$. The diagram is:

$$\begin{tikzcd}
N''\langle t'\rangle[y{\leftarrow}w'][x{\leftarrow}q] \arrow[r, multimap, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{com}"] & N''\langle u'\rangle[y{\leftarrow}w'][x{\leftarrow}q] \arrow[d, phantom, "\equiv_{com}"] \\
N''\langle t'\rangle[x{\leftarrow}q][y{\leftarrow}w'] \arrow[r, multimap, dashed, "\mathtt{lsv}"] & N''\langle u'\rangle[x{\leftarrow}q][y{\leftarrow}w']
\end{tikzcd}$$

(c) *Composition of substitutions* $\equiv_{[\cdot]}$. As in the previous case, there are two possibilities:
   i. *The substitution comes from* t'. That is, $N' = \langle\cdot\rangle$ and t' is a $\mapsto_{\mathtt{lsv}}$-redex $t' = N''\langle y\rangle[y{\leftarrow}L\langle v\rangle]$, with $x \notin \mathtt{fv}(N''\langle y\rangle)$. This case is exactly as the base exponential case 2(c)ii (read bottom-up).
   ii. *The substitution comes from* N'. That is, $N' = N''[y{\leftarrow}w']$ with $x \notin \mathtt{fv}(N''\langle t'\rangle)$. The diagram is:

$$\begin{tikzcd}
N''\langle t'\rangle[y{\leftarrow}w'][x{\leftarrow}q] \arrow[r, multimap] \arrow[d, phantom, "\equiv_{[\cdot]}"] & N''\langle u'\rangle[y{\leftarrow}w'][x{\leftarrow}q] \arrow[d, phantom, "\equiv_{[\cdot]}"] \\
N''\langle t'\rangle[y{\leftarrow}w'[x{\leftarrow}q]] \arrow[r, multimap, dashed] & N''\langle u'\rangle[y{\leftarrow}w'[x{\leftarrow}q]]
\end{tikzcd}$$

5. **Inductive case 3: inside a hereditary head substitution** $N = N'\langle x\rangle[x{\leftarrow}N'']$**.** The situation is:

$$t = N'\langle x\rangle[x{\leftarrow}N''\langle q\rangle] \multimap N'\langle x\rangle[x{\leftarrow}N''\langle q'\rangle] = u$$

If $\Leftrightarrow$ is internal to $N'\langle x\rangle$ the two steps clearly commutes. If $\Leftrightarrow$ is internal to $N''\langle q\rangle$ we conclude using the *i.h.*. The remaining cases are when $\Leftrightarrow$ overlaps with the topmost constructor. Axiom by axiom:

(a) *Commutation with the left of an application* $\equiv_{@l}$. It must be that $N'\langle x\rangle = N''''\langle x\rangle r$ with $x \notin \mathtt{fv}(r)$. Then the two steps simply commute:

$$\begin{tikzcd}
(N''''\langle x\rangle r)[x{\leftarrow}N''\langle q\rangle] \arrow[r, -o] \arrow[d, phantom, "\equiv_{@l}"] & (N''''\langle x\rangle r)[x{\leftarrow}N''\langle q'\rangle] \arrow[d, phantom, "\equiv_{@l}"] \\
N''''\langle x\rangle[x{\leftarrow}N''\langle q\rangle]r \arrow[r, dashed, -o] & N''''\langle x\rangle[x{\leftarrow}N''\langle q'\rangle]r
\end{tikzcd}$$

(b) *Commutation of independent substitutions* $\equiv_{com}$. It must be that $N'\langle x\rangle = N''''\langle x\rangle[y{\leftarrow}r]$ with $x \notin \mathtt{fv}(r)$. Then the two steps simply commute:

$$\begin{tikzcd}
N''''\langle x\rangle[y{\leftarrow}r][x{\leftarrow}N''\langle q\rangle] \arrow[r, -o] \arrow[d, phantom, "\equiv_{@l}"] & t_1 \arrow[d, phantom, "\equiv_{@l}"] \\
t_2 \arrow[r, dashed, -o] & t_3
\end{tikzcd}$$

where

$$t_1 := N''''\langle x\rangle[y{\leftarrow}r][x{\leftarrow}N''\langle q'\rangle],$$
$$t_2 := N''''\langle x\rangle[x{\leftarrow}N''\langle q\rangle][y{\leftarrow}r],$$
$$t_3 := N''''\langle x\rangle[x{\leftarrow}N''\langle q'\rangle][y{\leftarrow}r].$$

(c) *Composition of substitutions* $\equiv_{[\cdot]}$. There are various sub-cases

i. $[x{\leftarrow}N''\langle q\rangle]$ *enters in a substitution*. It must be that $N'\langle x\rangle = N_1\langle y\rangle[y{\leftarrow}N_2\langle x\rangle]$ with $x \notin \mathtt{fv}(N_1\langle y\rangle)$. Then the diagram is:

$$\begin{tikzcd}
N_1\langle y\rangle[y{\leftarrow}N_2\langle x\rangle][x{\leftarrow}N''\langle q\rangle] \arrow[r, -o] \arrow[d, phantom, "\equiv_{[\cdot]}"] & t_1 \arrow[d, phantom, "\equiv_{[\cdot]}"] \\
t_2 \arrow[r, dashed, -o] & t_3
\end{tikzcd}$$

$$t_1 := N_1\langle y\rangle[y{\leftarrow}N_2\langle x\rangle][x{\leftarrow}N''\langle q'\rangle],$$
$$t_2 := N_1\langle y\rangle[y{\leftarrow}N_2\langle x\rangle[x{\leftarrow}N''\langle q\rangle]],$$
$$t_3 := N_1\langle y\rangle[y{\leftarrow}N_2\langle x\rangle[x{\leftarrow}N''\langle q'\rangle]].$$

ii. *a substitution pops out of* $[x{\leftarrow}N''\langle q\rangle]$. Two sub-cases:

A. *The substitution comes from* N''. Then $N''\langle q\rangle = N''''\langle q\rangle[y{\leftarrow}r]$. The diagram is:

$$\begin{tikzcd}
N'\langle x\rangle[x{\leftarrow}N''''\langle q\rangle[y{\leftarrow}r]] \arrow[r, -o] \arrow[d, phantom, "\equiv_{[\cdot]}"] & t_1 \arrow[d, phantom, "\equiv_{[\cdot]}"] \\
t_2 \arrow[r, dashed, -o] & t_3
\end{tikzcd}$$

where

$$t_1 := N'\langle x\rangle[x{\leftarrow}N''''\langle q'\rangle[y{\leftarrow}r]], \tag{2}$$
$$t_2 := N'\langle x\rangle[x{\leftarrow}N''''\langle q\rangle][y{\leftarrow}r], \tag{3}$$
$$t_3 := N'\langle x\rangle[x{\leftarrow}N''''\langle q'\rangle][y{\leftarrow}r]. \tag{4}$$

B. *The substitution comes from* q. Then $N'' = \langle\cdot\rangle$ and q is a $\mapsto_{\mathtt{lsv}}$-redex $t' = N''''\langle y\rangle[y{\leftarrow}L\langle v\rangle]$ and the diagram is:

$$\begin{tikzcd}
N'\langle x\rangle[x{\leftarrow}N''''\langle y\rangle[y{\leftarrow}L\langle v\rangle]] \arrow[r, -o, "\mathtt{lsv}"] \arrow[d, phantom, "\equiv_{[\cdot]}"] & t_1 \arrow[d, phantom, "\equiv^*_{[\cdot]}"] \\
t_2 \arrow[r, dashed, -o, "\mathtt{lsv}"] & t_3
\end{tikzcd}$$

where

$$t_1 := N'\langle x\rangle[x{\leftarrow}L\langle N''''\langle v\rangle[y{\leftarrow}v]\rangle], \tag{5}$$
$$t_2 := N'\langle x\rangle[x{\leftarrow}N''''\langle y\rangle][y{\leftarrow}L\langle v\rangle], \tag{6}$$
$$t_3 := L\langle N'\langle x\rangle[x{\leftarrow}N''''\langle v\rangle][y{\leftarrow}v]\rangle. \tag{7}$$

B.5 Proofs for the LAM

Invariants, Lemma 6.3. By induction on the length of the execution leading to s, and straightforward inspection of the transition rules. □ □

Distillation, Theorem 6.4. 1. **Commutative 1.** We have $\overline{t}\overline{u} \mid e \mid \pi \to_{c_1} \overline{u} \mid e \mid \mathbf{f}(\overline{t}, e) :: \pi$, and:

$$\underline{\overline{t}\overline{u} \mid e \mid \pi} = \underline{\pi}\langle\underline{e}\langle\overline{t}\overline{u}\rangle\rangle \equiv^*_{@} \underline{\pi}\langle\underline{e}\langle\overline{t}\rangle\underline{e}\langle\overline{u}\rangle\rangle = \underline{\overline{u} \mid e \mid \mathbf{f}(\overline{t}, e) :: \pi}$$

As before, we use that π is a right-to-left call-by-value evaluation context, which enables us to use the $\equiv_{@}$ rule.

2. **Commutative 2.** We have $\overline{v} \mid e \mid \mathbf{f}(\overline{t}, e') :: \pi \to_{c_2} \overline{t} \mid e' \mid \mathbf{a}(\overline{v}, e) :: \pi$, and:

$$\underline{\overline{v} \mid e \mid \mathbf{f}(\overline{t}, e') :: \pi} = \underline{\pi}\langle\underline{e'}\langle\overline{t}\rangle\underline{e}\langle\overline{v}\rangle\rangle = \underline{\overline{t} \mid e' \mid \mathbf{a}(\overline{v}, e) :: \pi}$$

3. **Multiplicative.** We have $\lambda x.\overline{t} \mid e \mid \mathbf{a}(c) :: \pi \to_m \overline{t} \mid [x{\leftarrow}c] :: e \mid \pi$, and:

$$\underline{\lambda x.\overline{t} \mid e \mid \mathbf{a}(c) :: \pi} = \underline{\pi}\langle\underline{e}\langle\lambda x.\overline{t}\rangle\underline{c}\rangle \multimap_{\mathtt{m}} \underline{\pi}\langle\underline{e}\langle\overline{t}[x{\leftarrow}\underline{c}]\rangle\rangle$$

which is equal to $\underline{\overline{t} \mid [x{\leftarrow}c] :: e \mid \pi}$.

4. **Exponential.** Let $e = e'' :: [x{\leftarrow}(\overline{t}, e')] :: e'''$. We have $x \mid e \mid \pi \to_e \overline{t} \mid e' \mid \pi$, and::

$$\underline{x \mid e \mid \pi} = \underline{\pi}\langle\underline{e}\langle x\rangle\rangle \multimap_{\mathtt{e}} \underline{\pi}\langle\underline{e'''}\langle\underline{e'}\langle\underline{e''}\langle\overline{t}\rangle[x{\leftarrow}\overline{t}]\rangle\rangle\rangle \equiv^*_{gc} \underline{\pi}\langle\underline{e'}\langle\overline{t}\rangle\rangle =$$

Note that by Lemma 6.3.3, $\overline{t}$ is an abstraction, and thus we are able to apply $\multimap_{\mathtt{e}}$. Moreover, by Lemma 6.3.1, e binds variables to closures, and $\underline{e'}\langle\overline{t}\rangle$ is closed; this allows $\underline{e''}$ and $\underline{e'''}$ to be garbage collected. For doing so, the $\equiv_{gc}$ rule must be applied below a right-to-left call-by-value evaluation context, which follows from Lemma 6.3.4.

Progress. Let $s = \overline{t} \mid e \mid \pi$ be a commutative normal form s.t. $\underline{s} \multimap u$. If $\overline{t}$ is

- *an application* $\overline{u}\overline{w}$. Then a $\to_{c_1}$ transition applies and s is not a commutative normal form, absurd.
- *an abstraction* $\lambda x.\overline{u}$. Then $\underline{s} = \underline{\pi}\langle\underline{e}\langle\lambda x.\overline{u}\rangle\rangle$ is not in normal form. There can only be a $\multimap_{\mathtt{m}}$-redex, so $\underline{\pi}$ must be of the form $\underline{\pi}\langle\langle\cdot\rangle c\rangle$. This implies there is a $\to_m$ transition from s. □
- *a variable* x. Then $\underline{s} = \underline{\pi}\langle\underline{e}\langle x\rangle\rangle$ is not in normal form. There can only be a $\multimap_{\mathtt{e}}$-redex, and it must involve x, thus $\underline{e} = \underline{e'''}\langle\underline{e''}[x{\leftarrow}\underline{e'}\langle\overline{v}\rangle]\rangle$. This implies there is a $\to_e$ transition from s.

B.6 Proofs for the MAM

Proof. Let $\Leftrightarrow$ be the symmetric and contextual closure of the $\sim$ rule by which $\equiv_{\text{MAM}}$ is defined. Note $\equiv_{\text{MAM}}$ is the reflexive–transitive closure of $\Leftrightarrow$. It suffices to show that the property holds for $\Leftrightarrow$, *i.e.* that $w \Leftrightarrow\multimap u$ implies $w \multimap\equiv_{\text{MAM}} u$. The fact that $\Leftrightarrow^*$ is a bisimulation then follows by induction on the number of $\Leftrightarrow$ steps.

Let $w \Leftrightarrow t \multimap u$. The proof of $w \multimap\equiv_{\text{MAM}} u$ goes by induction on the call-by-need context N under which the $\multimap$-redex in t is contracted. Note that since $t_1 \sim t_2$ determines a bijection between the redexes of t_1 and t_2, it suffices to check the cases when $\sim$ is applied from left to right (*i.e.* $t \sim w$). For the right-to-left cases, all diagrams can be considered from bottom to top.

- **Base case, *i.e.* empty context** $N = \langle\cdot\rangle$**.** Two cases, depending on the $\multimap$ step contracting a $\mapsto_{\mathtt{dB}}$ or a $\mapsto_{\mathtt{ls}}$ redex:
  1. **Case** $t = L\langle\lambda x.t'\rangle u' \mapsto_{\mathtt{dB}} L\langle t'[x\leftarrow u']\rangle$**.** There are no $\sim$ redexes in t, since any application in t must be either t itself or below λx, which is not a call-by-name evaluation context.
  2. **Case** $t = N\langle x\rangle[x\leftarrow t'] \mapsto_{\mathtt{ls}} N\langle t'\rangle[x\leftarrow t']$**.** Any $\sim$ redex must be internal to N, in the sense that $N = N'\langle (N''u')[y\leftarrow w']\rangle$ with $y \notin \mathtt{fv}(u')$. Let $\widehat{N} = N'\langle N''[y\leftarrow w']u'\rangle$. Then:

$$\begin{array}{ccc} N\langle x\rangle[x\leftarrow t'] & \xrightarrow{\mathtt{ls}} & N\langle t'\rangle[x\leftarrow t'] \\ \sim & & \sim \\ \widehat{N}\langle x\rangle[x\leftarrow t'] & \overset{\mathtt{ls}}{\dashrightarrow} & \widehat{N}\langle t'\rangle[x\leftarrow t'] \end{array}$$

- **Inductive case** $N = N'q$**.** Since the application of $\Leftrightarrow$ must be internal to N', the result follows directly by *i.h.*.
- **Inductive case** $N = N'[x\leftarrow q]$**.** If the $\Leftrightarrow$ step is internal to N', the result follows again by applying *i.h.*. The remaining possibility is that $N'\langle t\rangle$ is an application. Here there are two cases:
  1. $N' = \langle\cdot\rangle$**, *i.e.* $\sim$ interacts with a redex.** The redex in question must be a dB-redex, since it must have an application at the root. The situation is the following, with $x \notin \mathtt{fv}(u')$:

$$\begin{array}{ccc} (L\langle\lambda y.t'\rangle\, u')[x\leftarrow q] & \xrightarrow{\mathtt{dB}} & L\langle t'[y\leftarrow u']\rangle[x\leftarrow q] \\ \sim & & = \\ L\langle\lambda y.t'\rangle[x\leftarrow q]\, u' & \overset{\mathtt{dB}}{\dashrightarrow} & L\langle t'[y\leftarrow u']\rangle[x\leftarrow q] \end{array}$$

  2. $N' = N''t'$**, *i.e.* there is no interaction between $\sim$ and a redex.** This case is straightforward, since the contraction of the $\multimap$ redex and the application of $\sim$ are orthogonal. $\square$

B.7 Proofs for the Split CEK

Split CEK Distillation, Theorem 7.7. Properties of the decoding:

1. *Commutative 1*. We have $\overline{t}\,\overline{u} \mid e \mid \pi \mid D \to_{c_1} \overline{t} \mid e \mid (\overline{u}, e) :: \pi \mid D$, and:

$$\begin{array}{lll} \underline{\overline{t}\,\overline{u} \mid e \mid \pi \mid D} & = & \underline{D}\langle\underline{\pi}\langle\underline{e}\langle\overline{t}\,\overline{u}\rangle\rangle\rangle & \equiv_@^* \\ & & \underline{D}\langle\underline{\pi}\langle\underline{e}\langle\overline{t}\rangle\,\underline{e}\langle\overline{u}\rangle\rangle\rangle & = \\ & & \underline{\overline{t} \mid e \mid (\overline{u}, e) :: \pi \mid D} \end{array}$$

2. *Commutative 2*. We have $\overline{v} \mid e \mid (\overline{t}, e') :: \pi \mid D \to_{c_2} \overline{t} \mid e' \mid \epsilon \mid ((\overline{v}, e), \pi) :: D$, and:

$$\begin{array}{llll} \underline{\overline{v} \mid e \mid (\overline{t}, e') :: \pi \mid D} & = & \underline{D}\langle\underline{\pi}\langle\underline{e}\langle v\rangle\,\underline{e'}\langle\overline{t}\rangle\rangle\rangle & \equiv_{gc}^* \\ & & \underline{D}\langle\underline{\pi}\langle\underline{e}\langle v\rangle\,\underline{e}\langle\underline{e'}\langle\overline{t}\rangle\rangle\rangle\rangle & \equiv_@^* \\ & & \underline{D}\langle\underline{\pi}\langle\underline{e}\langle v\,\underline{e'}\langle\overline{t}\rangle\rangle\rangle\rangle & = \\ & & \underline{\overline{t} \mid e' \mid \epsilon \mid ((\overline{v}, e), \pi) :: D} \end{array}$$

3. *Multiplicative*. We have $\overline{v} \mid e \mid \epsilon \mid ((\lambda x.\overline{t}, e'), \pi) :: D \to_m \overline{t} \mid [x\leftarrow(\overline{v}, e)] :: e' \mid \pi \mid D$, and:

$$\begin{array}{ll} \underline{\overline{v} \mid e \mid \epsilon \mid ((\lambda x.\overline{t}, e'), \pi) :: D} & = \\ \underline{D}\langle\underline{\pi}\langle\underline{e'}\langle(\lambda x.t)\,\underline{e}\langle v\rangle\rangle\rangle\rangle & \to_m \\ \underline{D}\langle\underline{\pi}\langle\underline{e'}\langle t[x\leftarrow\underline{e}\langle v\rangle]\rangle\rangle\rangle & = \\ \underline{\overline{t} \mid [x\leftarrow(\overline{v}, e)] :: e' \mid \pi \mid D} \end{array}$$

4. *Exponential*. We have $x \mid e_1 :: [x\leftarrow(\overline{v}, e)] :: e_2 \mid \pi \mid D \to_e \overline{v} \mid e \mid \pi \mid D$, and:

$$\begin{array}{ll} \underline{x \mid e_1 :: [x\leftarrow(\overline{v}, e)] :: e_2 \mid \pi \mid D} & = \\ \underline{D}\langle\underline{\pi}\langle\underline{e_2}\langle\underline{e_1}\langle x\rangle[x\leftarrow\underline{e}\langle\overline{v}\rangle]\rangle\rangle\rangle & \to_e \\ \underline{D}\langle\underline{\pi}\langle\underline{e_2}\langle\underline{e}\langle\underline{e_1}\langle\overline{v}\rangle[x\leftarrow\overline{v}]\rangle\rangle\rangle\rangle & \equiv_{gc}^* \\ \underline{D}\langle\underline{\pi}\langle\underline{e}\langle\overline{v}\rangle\rangle\rangle & = \\ \underline{\overline{v} \mid e \mid \pi \mid D} \end{array}$$

We use that $\underline{e}\langle\overline{v}\rangle$ is closed by Lemma 7.6.1 to ensure that $\underline{e_1}$, $\underline{e_2}$, and $[x\leftarrow\overline{v}]$ can be garbage collected.

Progress. Let $s = \overline{t} \mid e \mid \pi$ be a commutative normal form s.t. $\underline{s} \multimap u$. If $\overline{t}$ is

- *an application* $\overline{uw}$. Then a $\to_{c_1}$ transition applies and s is not a commutative normal form, absurd.
- *an abstraction* v. The decoding $\underline{s} = \underline{D}\langle\underline{\pi}\langle\underline{e}\langle\overline{v}\rangle\rangle\rangle$ must have a multiplicative redex, because it must have a redex and $\overline{v}$ is not a variable. So $\overline{v}$ is applied to something, *i.e.* there must be at least one application node in $\underline{D}\langle\underline{\pi}\rangle$. Moreover, the stack π must be empty, otherwise there would be an administrative $\to_{c_2}$ transition, contradicting the hypothesis. So D is not empty. Let $D = ((\overline{u}, e'), \pi') :: D'$. By point 3 of Lemma 7.6, $\overline{u}$ must be a value, and a $\to_m$ transition applies.
- *a variable* x. By point 1 of Lemma 7.6, x must be bound by e, so $e = e_1 :: [x\leftarrow(\overline{u}, e')] :: e_2$ and a $\to_e$ transition applies.

$\square$

B.8 Proofs for the Merged WAM

Distillation, Theorem 8.5. 1. *Commutative 1*. We have $\overline{t}\,\overline{u} \mid \pi \mid E \to_{c_1} \overline{t} \mid \mathbf{a}(\overline{u}) :: \pi \mid E$, and:

$$\underline{\overline{t}\,\overline{u} \mid \pi \mid E} = \underline{E}\langle\underline{\pi}\langle\overline{t}\,\overline{u}\rangle\rangle = \underline{\overline{t} \mid \mathbf{a}(\overline{u}) :: \pi \mid E}$$

2. *Commutative 2*. We have $x \mid \pi \mid E_1 :: [x\leftarrow\overline{t}] :: E_2 \to_{c_2} \overline{t} \mid \mathbf{h}(E_1, x) :: \pi \mid E_2$, and:

$$\begin{array}{llll} \underline{x \mid \pi \mid E_1 :: [x\leftarrow\overline{t}] :: E_2} & = & \underline{E_2}\langle\underline{E_1}\langle\underline{\pi}\langle x\rangle\rangle[x\leftarrow\overline{t}]\rangle & = \\ & & \underline{\overline{t} \mid \mathbf{h}(E_1, x) :: \pi \mid E_2} \end{array}$$

3. *Multiplicative*. We have $\lambda x.\overline{t} \mid \mathbf{a}(\overline{u}) :: \pi \mid E \to_m \overline{t} \mid \pi \mid [x\leftarrow\overline{u}] :: E$, and:

$$\begin{array}{llll} \underline{\lambda x.\overline{t} \mid \mathbf{a}(\overline{u}) :: \pi \mid E} & = & \underline{E}\langle\underline{\pi}\langle(\lambda x.\overline{t})\,\overline{u}\rangle\rangle & \multimap_{\mathtt{m}} \\ & & \underline{E}\langle\underline{\pi}\langle\overline{t}[x\leftarrow\overline{u}]\rangle\rangle & \equiv_{Lem.2.4} \\ & & \underline{E}\langle\underline{\pi}\langle\overline{t}\rangle[x\leftarrow\overline{u}]\rangle & = \\ & & \underline{\overline{t} \mid \pi \mid [x\leftarrow\overline{u}] :: E} \end{array}$$

4. *Exponential*. We have $\overline{v} \mid \mathbf{h}(E_1, x) :: \pi \mid E_2 \to_e \overline{v}^{\alpha} \mid \pi \mid E_1 :: [x\leftarrow\overline{v}] :: E_2$, and:

$$\begin{array}{llll} \underline{\overline{v} \mid \mathbf{h}(E_1, x) :: \pi \mid E_2} & = & \underline{E_2}\langle\underline{E_1}\langle\underline{\pi}\langle x\rangle\rangle[x\leftarrow\overline{v}]\rangle & \multimap_{\mathtt{e}} \\ & & \underline{E_2}\langle\underline{E_1}\langle\underline{\pi}\langle\overline{v}\rangle\rangle[x\leftarrow\overline{v}]\rangle & =_{\alpha} \\ & & \underline{E_2}\langle\underline{E_1}\langle\underline{\pi}\langle\overline{v}^{\alpha}\rangle\rangle[x\leftarrow\overline{v}]\rangle & = \\ & & \underline{\overline{v}^{\alpha} \mid \pi \mid E_1 :: [x\leftarrow\overline{v}] :: E_2} \end{array}$$

Progress. Let $s = \overline{t} \mid \pi \mid E$ be a commutative normal form s.t. $\underline{s} \multimap u$. If $\overline{t}$ is

1. *an application* $\overline{u}\overline{w}$. Then a $\to_{c_1}$ transition applies and s is not a commutative normal form, absurd.
2. *an abstraction* v. The decoding $\underline{s}$ is of the form $\underline{E}\langle\underline{\pi}\langle v\rangle\rangle$. The stack π cannot be empty, since then $\underline{s} = \underline{E}\langle v\rangle$ would be normal. So either the a $\to_e$ or a $\to_m$ transition applies.
3. *a variable* x. By the global closure invariant, x is bound by E. Then a $\to_{c_2}$ transition applies and s is not a commutative normal form, absurd. □

B.9 Proofs for the Pointing WAM

Pointing WAM Invariants, Lemma 9.3. By induction on the length of the execution. Points 1 and 2 are by direct inspection of the rules. Assuming $E \bot D$, point 4 is immediate by induction on the length of D.

Thus we are only left to check point 3. We use point 2, *i.e.* that substitutions in E bind pairwise distinct variables. Following we show that transitions preserve the invariant:

1. *Conmutative 1*. We have:
$$\overline{t}\,\overline{u} \mid \pi \mid D \mid E \to_{c_1} \overline{t} \mid \overline{u} :: \pi \mid D \mid E$$
Trivial, since the dump and the environment are the same and $\underline{(\overline{u} :: \pi)}\langle\overline{t}\rangle = \underline{\pi}\langle\overline{t}\,\overline{u}\rangle$.
2. *Conmutative 2*. We have $s \to_{c_2} s'$ with:
$$s = x \mid \pi \mid D \mid E_1 :: [x{\leftarrow}\overline{t}] :: E_2$$
$$s' = \overline{t} \mid \epsilon \mid (x,\pi) :: D \mid E_1 :: [x{\leftarrow}\Box] :: E_2$$
Note that since by *i.h.* $(\underline{\pi}\langle x\rangle, (E_1 :: [x{\leftarrow}\overline{t}] :: E_2){\upharpoonleft})$ is closed and x is free in $\underline{\pi}\langle x\rangle$, there cannot be any dumped substitutions in E_2. Then $(E_1 :: [x{\leftarrow}t] :: E_2){\upharpoonleft} = E_1{\upharpoonleft} :: [x{\leftarrow}t] :: E_2$ and we know:
$$(\underline{\pi}\langle x\rangle, E_1{\upharpoonleft} :: [x{\leftarrow}\overline{t}] :: E_2) \text{ is closed} \tag{8}$$
For 3a, note $(E_1 :: [x{\leftarrow}\Box] :: E_2){\upharpoonleft} = E_2$. Then we must show $(\overline{t}, E_2)$ is closed, which is implied by (8).

   For 3b, there are two cases:
   - If the pair is (x,π), we must show
   $$(\underline{\pi}\langle x\rangle, (E_1 :: [x{\leftarrow}\Box] :: E_2){\upharpoonleft}_x) \text{ is closed, } i.e.$$
   $$(\underline{\pi}\langle x\rangle, E_1{\upharpoonleft} :: [x{\leftarrow}\Box] :: E_2) \text{ is closed}$$
   which is implied by (8).
   - If the pair is (y,π') in D, with $y \neq x$, note first that
   $$(E_1 :: [x{\leftarrow}\overline{t}] :: E_2){\upharpoonleft}_y = E_1{\upharpoonleft}_y :: [x{\leftarrow}\overline{t}] :: E_2$$
   And similarly for $(E_1 :: [x{\leftarrow}\Box] :: E_2){\upharpoonleft}_y$. Moreover, by the invariant on s we know
   $$(\underline{\pi'}\langle y\rangle, E_1{\upharpoonleft}_y :: [x{\leftarrow}\overline{t}] :: E_2) \text{ is closed}$$
   and this implies
   $$(\underline{\pi'}\langle y\rangle, E_1{\upharpoonleft}_y :: [x{\leftarrow}\Box] :: E_2) \text{ is closed}$$
   as required.

   For 3c, we have already observed that E_2 has no dumped substitutions. Then $[x{\leftarrow}\Box]$ is the rightmost dumped substitution in the environment of s', while (x,π) is the leftmost pair in the dump. We conclude by the fact that the invariant already holds for s.
3. *Multiplicative, empty dump*. We have $s \to_m s'$ with:
$$s = \lambda x.\overline{t} \mid \overline{u} :: \pi \mid \epsilon \mid E$$
$$s' = \overline{t} \mid \pi \mid \epsilon \mid [x{\leftarrow}\overline{u}] :: E$$
First note that, since the environment and the dump are dual in s, there are no dumped substitutions in E.

   For point 3a, we know that:
$$(\underline{\pi}\langle(\lambda x.\overline{t})\,\overline{u}\rangle, E) \text{ is closed} \tag{9}$$
and we have to check:
$$(\underline{\pi}\langle\overline{t}\rangle, [x{\leftarrow}\overline{u}] :: E) \text{ is closed}$$
Let $y \in \mathtt{fv}(\underline{\pi}\langle\overline{t}\rangle)$. Then either $y = x$, which is bound by $[x{\leftarrow}\overline{u}]$, or $y \in \mathtt{fv}(\underline{\pi}\langle\lambda x.\overline{t}\rangle)$, in which case y is bound by E. Moreover, since $\underline{\pi}$ is an application context, by (9) we get $(\overline{u}, E)$ is closed.

   Points 3b and 3c are trivial since the dump is empty and the environment has no dumped substitutions.
4. *Multiplicative, non-empty dump*. We have $s \to_m s'$ with:
$$s = \lambda x.\overline{t} \mid \overline{u} :: \pi \mid (y,\pi') :: D \mid E_1 :: [y{\leftarrow}\Box] :: E_2$$
$$s' = \overline{t} \mid \pi \mid (y,\pi') :: D \mid E_1 :: [y{\leftarrow}\Box] :: [x{\leftarrow}\overline{u}] :: E_2$$
Note first that since the invariant holds for s, we know $[y{\leftarrow}\Box]$ is the rightmost dumped substitution in the environment of both s and s'. Therefore $(E_1 :: [y{\leftarrow}\Box] :: E_2){\upharpoonleft} = E_2$

   For proving point 3a, we have:
$$(\underline{\pi}\langle(\lambda x.\overline{t})\,\overline{u}\rangle, E_2) \text{ is closed}$$
and we must show:
$$(\underline{\pi}\langle\overline{t}\rangle, [x{\leftarrow}\overline{u}] :: E_2) \text{ is closed}$$
The situation is exactly as in point 3a for the $\to_m$ transition, empty dump case.

   For point 3b, let (z,π'') be any pair in $(y,\pi') :: D$. Let also
$$E_1' := \begin{cases} E_1{\upharpoonleft} & \text{if } y = z \\ E_1{\upharpoonleft}_z & \text{otherwise} \end{cases}$$
and note that $(E_1 :: [y{\leftarrow}\Box] :: E){\upharpoonleft}_y = E_1' :: [y{\leftarrow}\Box] :: E$ for any environment E that contains no dumped substitutions. By the invariant on s, we have that:
$$(\underline{\pi''}\langle z\rangle, E_1' :: [y{\leftarrow}\Box] :: E_2) \text{ is closed}$$
Moreover, from point 3a we know $(\overline{u}, E_2)$ is closed. Both imply:
$$(\underline{\pi''}\langle z\rangle, E_1' :: [y{\leftarrow}\Box] :: [x{\leftarrow}\overline{u}] :: E_2) \text{ is closed}$$
as required.

   For point 3c, just note that the substitution $[x{\leftarrow}\overline{u}]$ added to the environment is not dumped, and so duality holds because it holds for s by *i.h.*.
5. *Exponential*. We have $s \to_e s'$ with:
$$s = \overline{v} \mid \epsilon \mid (x,\pi) :: D \mid E_1 :: [x{\leftarrow}\Box] :: E_2$$
$$s' = \overline{v}^{\alpha} \mid \pi \mid D \mid E_1 :: [x{\leftarrow}\overline{v}] :: E_2$$
First note that since the environment and the dump are dual in s, we know E_2 has no dumped substitutions.

   For proving point 3a, by resorting to point 3a on the state s, for which the invariant already holds, we have that:
$$(\overline{v}, E_2) \text{ is closed} \tag{10}$$
Moreover, by point 3b on s, specialized on the pair (x,π), we also know:
$$(\underline{\pi}\langle x\rangle, E_1{\upharpoonleft} :: [x{\leftarrow}\Box] :: E_2) \text{ is closed} \tag{11}$$
We must check that:
$$(\underline{\pi}\langle\overline{v}^{\alpha}\rangle, E_1{\upharpoonleft} :: [x{\leftarrow}\overline{v}] :: E_2) \text{ is closed}$$

Any free variable in $\underline{\pi}\langle v^\alpha\rangle$ is either free in π, in which case by (10) it must be bound by $E_1 \upharpoonright :: [x \leftarrow \square] :: E_2$, or free in $\overline{v}$, in which case by (10) it must be bound by E_2. In both cases it is bound by $E_1 \upharpoonright :: [x \leftarrow \overline{v}] :: E_2$, as required. To conclude the proof of point 3a, note that by combining (10) and (11) we get $E_1 \upharpoonright :: [x \leftarrow v] :: E_2$ is closed.

For proving point 3b, let (y, π') be a pair in D. Using that $x \neq y$, by the invariant on s we know:

$$(\underline{\pi'}\langle y\rangle, E_1 \upharpoonright_y :: [x \leftarrow \square] :: E_2) \text{ is closed}$$

and this implies:

$$(\underline{\pi'}\langle y\rangle, E_1 \upharpoonright_y :: [x \leftarrow \overline{v}] :: E_2) \text{ is closed}$$

as wanted.

Point 3c is immediate, given that the environment and the dump are already dual in s.

□

B.10 Proofs for Distillation is Complexity Preserving

Theorem 10.3. 1. *LAM*. As for the CEK, using the corresponding subterm invariant and the following measure:

$$\#(\overline{u} \mid e \mid \pi) := \begin{cases} |\overline{u}| + |\overline{w}| & \text{if } \pi = \mathbf{f}(\overline{w}, e') :: \pi' \\ |\overline{u}| & \text{otherwise} \end{cases}$$

2. *Split CEK*. As for the CEK, using the corresponding subterm invariant and the following measure:

$$\#(\overline{u} \mid e \mid \pi \mid D) := \begin{cases} |\overline{u}| + |\overline{w}| & \text{if } \pi = (\overline{w}, e') :: \pi' \\ |\overline{u}| & \text{otherwise} \end{cases}$$

□