Logical Methods in Computer Science
Volume 20, Issue 3, 2024, pp. 3:1–3:45
https://lmcs.episciences.org/

Submitted May 17, 2022
Published Jul. 03, 2024

FULLY ABSTRACT ENCODINGS OF λ-CALCULUS IN HOcore THROUGH ABSTRACT MACHINES

MAŁGORZATA BIERNACKA [a], DARIUSZ BIERNACKI [a], SERGUEÏ LENGLET [b], PIOTR POLESIUK [a], DAMIEN POUS [c], AND ALAN SCHMITT [d]

[a] University of Wrocław
e-mail address: mabi@cs.uni.wroc.pl, dabi@cs.uni.wroc.pl, ppolesiuk@cs.uni.wroc.pl

[b] Université de Lorraine

[c] Plume, LIP, CNRS, ENS de Lyon, Université de Lyon
e-mail address: Damien.Pous@ens-lyon.fr

[d] INRIA
e-mail address: alan.schmitt@inria.fr

Abstract. We present fully abstract encodings of the call-by-name and call-by-value λ-calculus into HOcore, a minimal higher-order process calculus with no name restriction. We consider several equivalences on the λ-calculus side—normal-form bisimilarity, applicative bisimilarity, and contextual equivalence—that we internalize into abstract machines in order to prove full abstraction of the encodings. We also demonstrate that this technique scales to the $\lambda\mu$-calculus, i.e., a standard extension of the λ-calculus with control operators.

1. Introduction

HOcore is a minimal process calculus with higher-order communication, meaning that messages are executable processes. It is a subcalculus of HOπ [San96] with no construct to generate names or to restrict the scope of communication channels. Even with such a limited syntax, HOcore is Turing complete [LPSS11]. However, as a higher-order calculus, it is less expressive than the name passing π-calculus: polyadic message sending cannot be compositionally encoded in monadic message sending in HOπ [LPSS10], while it can be done in π [SW01].

Although HOcore is Turing complete, a fully abstract encoding of the λ-calculus into HOcore appears impossible at first. Indeed, a λ-term potentially has an unbounded number of redexes. A straightforward encoding would use communication to emulate β-reduction, but since HOcore does not provide means to restrict the scope of communication, one would need as many distinct names as there are redexes to avoid interference. Moreover, as new redexes may be created by β-reduction, we also need a way to generate new names on which to communicate. To circumvent these problems and illustrate the expressiveness

Key words and phrases: lambda calculus, process calculi, abstract machine, control operator, bisimilarity, full abstraction.

LOGICAL METHODS IN COMPUTER SCIENCE DOI:10.46298/LMCS-20(3:3)2024

of HOcore, we consider encodings where the *reduction strategy* is fixed, thus for which at most one redex is enabled at any time. In this setting, β-reduction can be emulated using communication on a single, shared, name. A first contribution of this paper is the definition of novel encodings of the call-by-name and call-by-value λ-calculus—more precisely the Krivine Abstract Machine (KAM) [Kri07] and CK machine [FFF09]—into HOcore.

A faithful encoding not only reflects the operational semantics of a calculus, it should also reflect its equivalences. Ideally, an encoding is *fully abstract*: two source terms are behaviorally equivalent iff their translations are equivalent. On the HOcore side, we use *barbed equivalence with hidden names* [LPSS11], where a fixed number of names used for the translation cannot be observed. On the λ-calculus side, we consider three equivalences. First, we look at *normal-form bisimilarity* [Las99b], where normal forms are decomposed into subterms that must be bisimilar. Next, we turn to *applicative bisimilarity* [AO93], where normal forms must behave similarly when applied to identical arguments. And finally, we consider contextual equivalence, where terms must behave identically when put into arbitrary contexts.

Our second contribution is an *internalization* of these equivalences into extended abstract machines: these machines expand the underlying machine evaluating the term (the KAM or CK machine) with additional transitions with flags interpreting the equivalences. By doing so, we can express these different equivalences on terms by a simpler bisimilarity on the flag-generating machines, which can be seen as labeled transition systems (LTS). We then translate these extended machines into HOcore and prove full abstraction for all three equivalences. The result of the translation is therefore a HOcore process representing the source λ-term augmented with the machine of the internalized equivalence under consideration. A surrounding context may then interact with that process by communicating on the machine flags.

Finally, we show that the internalization principle scales to the $\lambda\mu$-calculus [Par92], an extension of the λ-calculus with control operators. The μ operator is able to capture the current continuation for later use. Its semantics can be defined with an extended KAM, in which we can internalize normal-form and applicative bisimilarities as well as contextual equivalence. Altogether, this work shows that a minimal process calculus with no name restriction can faithfully encode the call-by-name and call-by-value λ-calculus, with or without control operators.

The chosen equivalences. It is enough to faithfully encode contextual equivalence to get full abstraction. We choose to study other equivalences (normal-form and applicative bisimilarities) for two reasons. First, we start with normal-form bisimilarity because it is the simplest to translate, as we do not need to inject terms from the environment to establish the equivalence. We next show how we can inject terms for applicative bisimilarity, and we finally extend this approach to contexts for contextual equivalence. Second, the study of quite different equivalences illustrate the robustness of the internalization technique.

Related work. Since Milner's seminal work [Mil92], other encodings of λ into π have been proposed either as a result in itself [CF11, DHS22], or to study other properties such as connections with logic [Acc13, Bef11, TCP12], termination [CCS10, Ama11, YBH04], sequentiality [BHY01], control [CCS10, HYB14, vBV14], references [Pre22], or Continuation-Passing Style (CPS) transforms [San99, SW01, DMAV14]. These works use the more expressive *first-order* π-calculus, except for [San99, SW01], discussed below; full abstraction

is proved w.r.t. contextual equivalence in [BHY01, YBH04, HYB14], normal-form bisimilarity in [vBV14, DHS22, Pre22], applicative bisimilarity in [CF11], and both bisimilarities in [SW01]. The encodings of [BHY01, YBH04, HYB14] are driven by types, and therefore cannot be compared to our untyped setting. In [vBV14], van Bakel et al. establish a full abstraction result between the $\lambda\mu$-calculus with normal-form bisimilarity and the π-calculus. Their encoding relies on an unbounded number of restricted names to evaluate several translations of λ-terms in parallel, while we rely on flags and on barbed equivalence. We explain the differences between the two approaches in Section 4.2. The encoding of [CF11] also uses an unbounded number of restricted names, to represent a λ-term as a tree and to process it.

Sangiorgi translates the λ-calculus into a higher-order calculus as an intermediary step in [San99, SW01], but it is an abstraction-passing calculus, which is strictly more expressive than a process-passing calculus [LPSS10]. Like in our work, Sangiorgi fixes the evaluation strategy in the λ-calculus, except that he uses CPS translations rather than abstract machines. In the light of Danvy et al.'s functional correspondence [ABDM03], the two approaches appear closely related, however it is difficult to compare our encoding with Sangiorgi's, since we target different calculi, and we internalize the bisimilarities in the abstract machines. Still, name restriction plays an important role in Sangiorgi's encodings, since a local channel is used for each application in a λ-term. The encoding is fully abstract w.r.t. normal-form bisimilarity [SW01, Chapter 18] but not w.r.t. applicative bisimilarity [SW01, Chapter 17]. Indeed, a translated λ-abstraction waits for the outside to provide an access to an encoded argument to be applied to. However, the environment may give access to a random process and not to a translated λ-term. The encoding of [SW01] does not protect itself against this unwanted behavior from the environment. In contrast, the encoding of Section 5 and the one in [CF11] are fully abstract w.r.t. applicative bisimilarity, because they take this issue into account, as we explain in Section 5.2.

Outline. Section 2 describes HOcore. We present the main ideas behind our encodings in Section 3 on the KAM. We explain how to internalize equivalences into abstract machines and how to translate the resulting machines for three languages: call-by-name λ-calculus (Sections 4 and 5), call-by-value λ-calculus (Section 6), and call-by-name $\lambda\mu$-calculus (Section 7). For each of them, we discuss normal-form and applicative bisimilarities, and contextual equivalence. Section 8 concludes this paper.

Compared to the conference publication [BBL+17], Sections 6 and 7 are new, and the proofs have been inlined in Sections 4 and 5.

2. The Calculus HOcore

Syntax and semantics. HOcore [LPSS11] is a simpler version of HOπ [San96] where name restriction is removed. We let a, b, *etc.* range over channel names, and x, y, *etc.* range over process variables. The syntax of HOcore processes is as follows.

$$P, Q ::= a(x).P \mid \overline{a}\langle P\rangle \mid P \parallel Q \mid x \mid \mathbf{0}$$

The process $a(x).P$ is waiting for a message on a which, when received, is substituted for the variable x in P. If x does not occur in P, then we write $a(\_).P$. The process $\overline{a}\langle P\rangle$ is sending a message on a. Note that communication is higher order—processes are sent—and asynchronous—there is no continuation after a message output. The parallel composition

$$\textsc{Out}\ \frac{}{\overline{a}\langle P\rangle \xrightarrow{\overline{a}\langle P\rangle} \mathbf{0}} \qquad \textsc{Inp}\ \frac{}{a(x).Q \xrightarrow{a(P)} [P\,/\,x]Q}$$

$$\textsc{Par}\ \frac{P \xrightarrow{l} P'}{P \parallel Q \xrightarrow{l} P' \parallel Q} \qquad \textsc{Tau}\ \frac{P \xrightarrow{\overline{a}\langle R\rangle} P' \qquad Q \xrightarrow{a(R)} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

Figure 1: HOcore LTS

of processes is written $P \parallel Q$, and the process $\mathbf{0}$ cannot perform any action. Input has a higher precedence than parallel composition, e.g., we write $a(x).P \parallel Q$ for $(a(x).P) \parallel Q$. We implicitly consider that parallel composition is associative, commutative, and has $\mathbf{0}$ as a neutral element. In an input $a(x).P$, the variable x is bound in P. We assume bound names to be pairwise distinct and distinct from free names, using α-conversion if necessary; we follow this convention for all binding constructs in this paper. We write $\mathsf{fn}(P)$ for the free channel names of P.

Informally, when an output $\overline{a}\langle P\rangle$ is in parallel with an input $a(x).Q$, a communication on a takes place, producing $[P\,/\,x]Q$, the capture avoiding substitution of x by P in Q. We define in Figure 1 the semantics of HOcore as a LTS, omitting the rules symmetric to Par and Tau. The labels (ranged over by l) are either τ for internal communication, $\overline{a}\langle P\rangle$ for message output, or $a(P)$ for process input. We label a an input where the received process does not matter (e.g., $a(\_).P \xrightarrow{a} P$).

Weak transitions allow for internal actions before and after a visible one. We write $\xRightarrow{\tau}$ for the reflexive and transitive closure $\xrightarrow{\tau}{}^{*}$, and $\xRightarrow{l}$ for $\xRightarrow{\tau}\xrightarrow{l}\xRightarrow{\tau}$ when $l \neq \tau$.

Barbed equivalence. Our definition of observable action (or barb) and barbed equivalence depends on a finite set $\mathbb{H}$ of hidden names, which allows some observable actions to be ignored. Instead of adding top-level name restrictions on these names, as in [LPSS11], we prefer to preserve the semantics of the calculus and simply hide some names in the equivalence. Hidden names are not a computational construct and are not required for the encoding of the KAM, but they are necessary to protect the encoding from an arbitrary environment when proving full abstraction. We emphasize that we do not need the full power of name restriction: the set of hidden names is finite and static—there is no way to create new hidden names.

We let γ range over names a and *conames* $\overline{a}$.

Definition 2.1. The process P has a strong observable action on a (resp. $\overline{a}$) w.r.t. $\mathbb{H}$, written $P \downarrow_a^{\mathbb{H}}$ (resp. $P \downarrow_{\overline{a}}^{\mathbb{H}}$), if $a \notin \mathbb{H}$ and $P \xrightarrow{a(Q)} R$ (resp. $P \xrightarrow{\overline{a}\langle Q\rangle} R$) for some Q and R. A process P has a weak observable action on γ w.r.t. $\mathbb{H}$, written $P \Downarrow_\gamma^{\mathbb{H}}$, if $P \xRightarrow{\tau} P' \downarrow_\gamma^{\mathbb{H}}$ for some P'. We write $\mathsf{WkObs}_{\mathbb{H}}(P)$ for the set of weak observable actions of P w.r.t. $\mathbb{H}$.

Definition 2.2. A symmetric relation $\mathcal{R}$ is a barbed bisimulation w.r.t. $\mathbb{H}$ if $P\,\mathcal{R}\,Q$ implies
- $P \downarrow_\gamma^{\mathbb{H}}$ implies $Q \Downarrow_\gamma^{\mathbb{H}}$;
- for all R such that $\mathsf{fn}(R) \cap \mathbb{H} = \varnothing$, we have $P \| R \;\; \mathcal{R} \;\; Q \| R$;

• if $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \xRightarrow{\tau} Q'$ and $P' \mathcal{R} Q'$.

Barbed equivalence w.r.t. $\mathbb{H}$, noted $\approx^{\mathbb{H}}_{\mathsf{HO}}$, is the largest barbed bisimulation w.r.t. $\mathbb{H}$.

A strong barbed equivalence can be defined by replacing $\Downarrow^{\mathbb{H}}_{\gamma}$ with $\downarrow^{\mathbb{H}}_{\gamma}$ in the first item, and $\xRightarrow{\tau}$ with $\xrightarrow{\tau}$ in the third. From [LPSS11], we know that strong barbed equivalence is decidable when all names can be observed ($\mathbb{H} = \varnothing$), but undecidable with 4 hidden names. We lower this bound to 2 in Theorem 3.3.

3. Encoding the Krivine Abstract Machine

We show in this section that HOcore may faithfully encode a call-by-name λ-calculus through an operationally equivalent encoding of the KAM.

3.1. **Definition of the KAM.** The KAM [Kri07] is a machine for call-by-name evaluation of closed λ-calculus terms. We present a substitution-based variant of the KAM for simplicity, and to reuse the substitution of HOcore in the translation. A *configuration* C of the machine is composed of the term t being evaluated, and a stack π of λ-terms. Their syntax and the transitions are as follows.

$$C ::= \langle t, \pi \rangle \qquad \text{(configurations)}$$
$$t, s ::= x \mid t\ s \mid \lambda x.t \qquad \text{(terms)}$$
$$\pi ::= t :: \pi \mid [] \qquad \text{(stacks)}$$

$$\langle t\ s, \pi \rangle \mapsto \langle t, s :: \pi \rangle \qquad \text{(Push)}$$
$$\langle \lambda x.t, s :: \pi \rangle \mapsto \langle [s\,/\,x]t, \pi \rangle \qquad \text{(Grab)}$$

A λ-abstraction $\lambda x.t$ binds x in t; a term is closed if it does not contain any free variables. We use $[]$ to denote the empty stack. In Push, the argument s of an application is stored on the stack while the term t in function position is evaluated. If we get a λ-abstraction $\lambda x.t$, then an argument s is fetched from the stack (transition Grab), and the evaluation continues with $[s\,/\,x]t$, the capture-avoiding substitution of x by s in t. If a configuration of the form $\langle \lambda x.t, [] \rangle$ is reached, then the evaluation is finished, and the result is $\lambda x.t$. Because we evaluate closed terms only, it is not possible to obtain a configuration of the form $\langle x, \pi \rangle$.

3.2. **Translation into HOcore.** The translation of the KAM depends essentially on how we push and grab terms on the stack. We represent the stack by two messages, one on name hd_{c} for its head, and one on name c (for *continuation*) for its tail (henceforth, a stack q is always encoded as a message on hd_{q} for its head and one on q for its tail). The empty stack can be represented by an arbitrary, non-diverging, deterministic process, e.g., $\mathbf{0}$; here we use a third name to signal that the computation is finished with $\overline{b}\langle \mathbf{0} \rangle$. As an example, the stack $1 :: 2 :: 3 :: 4 :: []$ is represented by $\overline{hd_{\mathsf{c}}}\langle 1 \rangle \parallel \overline{c}\Big\langle \overline{hd_{\mathsf{c}}}\langle 2 \rangle \parallel \overline{c}\Big\langle \overline{hd_{\mathsf{c}}}\langle 3 \rangle \parallel \overline{c}\Big\langle \overline{hd_{\mathsf{c}}}\langle 4 \rangle \parallel \overline{c}\big\langle \overline{b}\langle \mathbf{0} \rangle \big\rangle \Big\rangle \Big\rangle \Big\rangle$.

With this representation, pushing an element e on a stack p is done by creating the process $\overline{hd_{\mathsf{c}}}\langle e \rangle \parallel \overline{c}\langle p \rangle$, while grabbing the head of the stack corresponds to receiving on hd_{c}.

With this idea in mind, we define the translations of the entities defining the KAM, starting with stacks.

$$[\![[]]\!] \triangleq \overline{b}\langle \mathbf{0} \rangle$$
$$[\![t :: \pi]\!] \triangleq \overline{hd_{\mathsf{c}}}\langle [\![t]\!] \rangle \parallel \overline{c}\langle [\![\pi]\!] \rangle$$

In the translation of a configuration $\langle t, \pi \rangle$, we reuse the name c to store the stack, meaning that before pushing on π or grabbing the head of π, we have to get $[\![\pi]\!]$ by receiving on c.

$$[\![\langle t, \pi \rangle]\!] \triangleq [\![t]\!] \parallel \overline{c}\langle [\![\pi]\!] \rangle$$

For instance, in the application case $[\![t\ s]\!]$, we start by receiving the current stack p on c, and we then run $[\![t]\!]$ in parallel with the translation of the new stack $\overline{hd_{\mathsf{c}}}\langle [\![s]\!] \rangle \parallel \overline{c}\langle p \rangle$. Henceforth, we assume the variable p does not occur free in the translated entities.

$$[\![t\ s]\!] \triangleq c(p).\big([\![t]\!] \parallel \overline{c}\big\langle \overline{hd_{\mathsf{c}}}\langle [\![s]\!] \rangle \parallel \overline{c}\langle p \rangle \big\rangle\big)$$

Similarly, in the λ-abstraction case $[\![\lambda x.t]\!]$, we get the current stack p on c, that we run in parallel with $hd_{\mathsf{c}}(x).[\![t]\!]$. If p is not empty, then it is a process of the form $\overline{hd_{\mathsf{c}}}\langle [\![s]\!] \rangle \parallel \overline{c}\langle [\![\pi]\!] \rangle$, and a communication on hd_{c} is possible, realizing the substitution of x by s in t; the execution then continues with $[\![[s\,/\,x]t]\!] \parallel \overline{c}\langle [\![\pi]\!] \rangle$. Otherwise, p is $\overline{b}\langle \mathbf{0} \rangle$, and the computation terminates.

$$[\![\lambda x.t]\!] \triangleq c(p).(hd_{\mathsf{c}}(x).[\![t]\!] \parallel p)$$
$$[\![x]\!] \triangleq x$$

Formally, the operational correspondence between the KAM and its translation is as follows.

Theorem 3.1. *In the forward direction, if $C \mapsto^* C'$, then $[\![C]\!] \stackrel{\tau}{\Longrightarrow} [\![C']\!]$. In the backward direction, if $[\![C]\!] \stackrel{\tau}{\Longrightarrow} P$, then there exists a C' such that $C \mapsto^* C'$ and either*

- $P = [\![C']\!]$,
- *or there exists P' such that $P \stackrel{\tau}{\longrightarrow} P' = [\![C']\!]$,*
- *or $C' = \langle \lambda x.t, [] \rangle$ and $P = hd_{\mathsf{c}}(x).[\![t]\!] \parallel \overline{b}\langle \mathbf{0} \rangle$.*

Proof. The proof is straightforward in the forward direction. In the backward direction, we show that the translation is deterministic (if $[\![C]\!] \stackrel{\tau}{\Longrightarrow} P \stackrel{\tau}{\longrightarrow} Q_1$ and $[\![C]\!] \stackrel{\tau}{\Longrightarrow} P \stackrel{\tau}{\longrightarrow} Q_2$, then $Q_1 = Q_2$) and we rely on the fact that the translation of a Push step uses one communication, while we use two communications for a Grab step. □

A direct consequence is we can observe on the HOcore side when the reduction of a λ-term converges.

Corollary 3.2. *$C \mapsto^* \langle \lambda x.t, [] \rangle$ for some t iff $[\![C]\!] \Downarrow_{\overline{b}}^{\varnothing}$.*

Proof. If $C \mapsto^* \langle \lambda x.t, [] \rangle$ for some t, then by Theorem 3.1, $[\![C]\!] \stackrel{\tau}{\Longrightarrow} [\![\langle \lambda x.t, [] \rangle]\!] \stackrel{\tau}{\longrightarrow} hd_{\mathsf{c}}(x).[\![t]\!] \parallel \overline{b}\langle \mathbf{0} \rangle$, and the result holds.

Conversely, suppose $[\![C]\!] \Downarrow_{\overline{b}}^{\varnothing}$: there exists P such that $[\![C]\!] \stackrel{\tau}{\Longrightarrow} P \downarrow_{\overline{b}}^{\varnothing}$. One can check that in the first two cases of the backward direction of Theorem 3.1, P cannot exhibit $\overline{b}$ as an observable action, so we are in the last case, as wished. □

We can then improve over the result of [LPSS11] about undecidability of strong barbed equivalence by hiding hd_{c} and c.

Theorem 3.3. *Strong barbed equivalence is undecidable in HOcore with 2 hidden names.*

Proof. Assume we can decide strong barbed equivalence with two hidden names. Let t be a closed λ-term and $P_\Omega \triangleq c(x).(x \parallel \overline{c}\langle x\rangle) \parallel \overline{c}\big(c(x).(x \parallel \overline{c}\langle x\rangle)\big)$. We can thus decide if $[\![\langle t, []\rangle]\!]$ is strong barbed-equivalent to P_Ω when hd_{c} and c are hidden, but P_Ω loops with no observable action, and t converges iff $[\![\langle t, []\rangle]\!]$ has a weak observable action on $\overline{b}$ by Corollary 3.2. By deciding barbed equivalence, we can therefore decide whether the reduction of t converges, hence a contradiction. □

4. Normal-Form Bisimilarity

Our first full abstraction result is for normal-form bisimilarity [Las99b]. We show how to internalize this equivalence in an extension of the KAM such that it may be captured by a simple barbed bisimilarity. We then translate this extended KAM into HOcore, and we finally prove full abstraction.

4.1. Normal-Form Bisimilarity.

Normal-form bisimilarity compares terms by reducing them to weak head normal forms, if they converge, and then decomposes these normal forms into subterms that must be bisimilar. Unlike the KAM, normal-form bisimilarity is defined on open terms, thus we distinguish free variables, ranged over by f, from bound variables, ranged over by x. The grammars of terms (t, s) and values (v) become as follows.

$$t, s ::= f \mid x \mid \lambda x.t \mid t\ s \qquad v ::= f \mid \lambda x.t$$

Henceforth, we assume that λ-terms are well formed, i.e., all variables ranged over by x are bound: x is not a valid term but f is. We write $\mathsf{fv}(t)$ for the set of free variables of t. A variable f is said *fresh* if it does not occur in any of the entities under consideration.

When evaluating an open term, we can obtain either a λ-abstraction with an empty stack, or a free variable and a stack. We inductively extend a relation $\mathcal{R}$ on λ-terms to stacks by writing $\pi_1\ \mathcal{R}\ \pi_2$ if $\pi_1 = \pi_2 = []$, or if $\pi_1 = t :: \pi_1'$, $\pi_2 = s :: \pi_2'$, $t\ \mathcal{R}\ s$, and $\pi_1'\ \mathcal{R}\ \pi_2'$.

Definition 4.1. A symmetric relation $\mathcal{R}$ is a normal-form bisimulation if $t\ \mathcal{R}\ s$ implies:

- if $\langle t, []\rangle \mapsto^* \langle \lambda x.t', []\rangle$, then there exists s' such that $\langle s, []\rangle \mapsto^* \langle \lambda x.s', []\rangle$ and $[f\,/\,x]t'\ \mathcal{R}\ [f\,/\,x]s'$ for a fresh f;
- if $\langle t, []\rangle \mapsto^* \langle f, \pi\rangle$, then there exists π' such that $\langle s, []\rangle \mapsto^* \langle f, \pi'\rangle$ and $\pi\ \mathcal{R}\ \pi'$.

Normal-form bisimilarity $\approx_{\mathsf{nf}}$ is the largest normal-form bisimulation.

Normal-form bisimilarity is not complete w.r.t. contextual equivalence in λ-calculus: there exists contextually equivalent terms which are not normal-form bisimilar [San92, Example 5.5]. In this work, we evaluate terms to weak-head normal forms, and the resulting bisimilarity characterizes Lévy-Longo tree equivalence [Lév75, Ong88, San92]. If we were to evaluate to head normal forms instead, it would characterize Böhm tree equivalence [Bar84, Las99b].

$$\langle t\ s, \pi, n\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto} \langle t, s :: \pi, n\rangle_{\mathsf{ev}} \qquad \text{(Push)}$$

$$\langle \lambda x.t, s :: \pi, n\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto} \langle [s\,/\,x]t, \pi, n\rangle_{\mathsf{ev}} \qquad \text{(Grab)}$$

$$\langle \lambda x.t, [], n\rangle_{\mathsf{ev}} \overset{\circledcirc}{\mapsto} \langle [n\,/\,x]t, [], n+1\rangle_{\mathsf{ev}} \qquad \text{(Lambda)}$$

$$\langle f, \pi, n\rangle_{\mathsf{ev}} \overset{f}{\mapsto} \langle \pi, n\rangle_{\mathsf{ct}} \qquad \text{(Var)}$$

$$\langle [], n\rangle_{\mathsf{ct}} \overset{\bigstar}{\mapsto} \qquad \text{(Done)}$$

$$\langle t :: \pi, n\rangle_{\mathsf{ct}} \overset{\blacktriangledown}{\mapsto} \langle t, [], n\rangle_{\mathsf{ev}} \qquad \text{(Enter)}$$

$$\langle t :: \pi, n\rangle_{\mathsf{ct}} \overset{\blacktriangleright}{\mapsto} \langle \pi, n\rangle_{\mathsf{ct}} \qquad \text{(Skip)}$$

Figure 2: NFB Machine

4.2. **Abstract Machine.** We extend the KAM so that it provides additional steps, identified by labeled transitions, that capture the testing done by normal-form bisimilarity. Roughly, the extended machine explores the Lévy-Longo tree of a λ-term, choosing arbitrarily a branch when several are available, but signaling these choices to the outside so that they can be mimicked. To do so, we rely on *flagged transitions*, *terminating transitions*, and a restricted form of *non-determinism*.

Flagged transitions are usual transitions of the machine with some additional information to convey to the environment that a particular event is taking place. Machine bisimilarity, defined below, ensures that bisimilar machines have matching flags. Transitions without flags use a τ label. Terminating transitions are flagged transitions that indicate the computation has stopped. They are needed for bisimilarity: as machine bisimilarity ignores τ labels, we use terminating transitions to distinguish between terminated and divergent machine runs. Finally, we allow some non-determinism in machines, i.e., a given configuration may take two different transitions, only if the transitions are flagged and have different flags. In other words, the non-deterministic choice is made explicit to the environment.

We define the NFB machine in Figure 2. When computation stops with a λ-abstraction and an empty stack, we have to restart the machine to evaluate the body of the abstraction with a freshly generated free variable (rule Lambda). To do so, we consider free variables as natural numbers, and we keep a counter n in the machine which is incremented each time a fresh variable is needed. For a configuration $\langle f, \pi\rangle$, normal-form bisimilarity evaluates each of the t_i in the stack (rule Var). To internalize this step, we could launch several machines in parallel, as in [vBV14], where the translated t_i are run in parallel. This approach has two drawbacks: first, it is a further extension of abstract machines—a machine no longer steps to a single machine state but to a multiset of states. Second, when translating such extended machines into HOcore, we want to prevent them from interacting with each other, but we cannot rely on name restriction, as in [vBV14], to encapsulate an unbounded number of translations. Alternatively, one could evaluate the elements of the stack sequentially, but this approach fails if one of the elements of the stack diverges, as the later elements will never be evaluated.

We thus consider a third approach, built upon flagged non-determinism: the machine chooses arbitrarily an element of the stack to evaluate, and signals this choice using flags (rules Enter, Skip, and Done). The burden of evaluating every element of the stack is

thus relegated to the definition of machine bisimilarity: as every flagged execution must be matched by an execution with the same flags, every possible choice is explored. It is reminiscent of how operational game semantics [LS14, Jab15] equates terms: the possible interactions of a program with its environment are represented with labeled transitions, the execution of a program is a trace of such labels, and trace equivalence ensures that every possible execution has been considered.

As before, we use C to range over configurations, which are now of two kinds. In *evaluation mode*, $\langle t, \pi, n\rangle_{\mathsf{ev}}$ is reducing t within stack π and with counter n. The transitions Push and Grab are as in the KAM, except for the extra parameter. If we reach a λ-abstraction in the empty context (transition Lambda), then the machine flags $\circledcirc$ and then restarts to evaluate the body, replacing the bound variable by a fresh free variable, i.e., the current value n of the counter. If we reach a free variable f, i.e., a number, then we flag f before entering the next mode (transition Var).

In *continuation mode* $\langle \pi, n\rangle_{\mathsf{ct}}$, the transition Done simply finishes the execution if $\pi = []$, using the flag $\star$. Otherwise, $\pi = t :: \pi'$, and the machine either evaluates t with flag $\blacktriangledown$ (and forgets about π'), or skips t with a flag $\blacktriangleright$ to possibly evaluate a term in π'. The machine may skip the evaluation of all the terms in π, but it would still provide some information, as it would generate m $\blacktriangleright$ messages (followed by $\star$), telling us that π has m elements. Note that the counter n is stored in continuation mode just to be passed to the evaluation mode when one of the t_i is chosen with transition Enter.

Example 4.2. To illustrate how the machine works, we show the transitions starting from the term $(\lambda x.x)\ (\lambda y.y\ 0\ \Omega)$, where $\Omega \triangleq (\lambda x.x\ x)\ (\lambda x.x\ x)$. The term is executed in the empty context, and with a counter initialized to a value greater than its free variables.

$$
\begin{aligned}
&\langle (\lambda x.x)\ (\lambda y.y\ 0\ \Omega), [], 1\rangle_{\mathsf{ev}} \\
&\overset{\tau}{\mapsto} \langle \lambda x.x, \lambda y.y\ 0\ \Omega :: [], 1\rangle_{\mathsf{ev}} && \text{(Push)} \\
&\overset{\tau}{\mapsto} \langle \lambda y.y\ 0\ \Omega, [], 1\rangle_{\mathsf{ev}} && \text{(Grab)} \\
&\overset{\circledcirc}{\mapsto} \langle 1\ 0\ \Omega, [], 2\rangle_{\mathsf{ev}} && \text{(Lambda)} \\
&\overset{\tau}{\mapsto} \langle 1\ 0, \Omega :: [], 2\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto} \langle 1, 0 :: \Omega :: [], 2\rangle_{\mathsf{ev}} && \text{(Push - Push)} \\
&\overset{1}{\mapsto} \langle 0 :: \Omega :: [], 2\rangle_{\mathsf{ct}} && \text{(Var)}
\end{aligned}
$$

We then have three possibilities. First, we reduce the top of the stack, with the sequence $\langle 0 :: \Omega :: [], 2\rangle_{\mathsf{ct}} \overset{\blacktriangledown}{\mapsto} \langle 0, [], 2\rangle_{\mathsf{ev}} \overset{0}{\mapsto} \langle [], 2\rangle_{\mathsf{ct}} \overset{\star}{\mapsto}$. Second, we evaluate Ω with the sequence $\langle 0 :: \Omega :: [], 2\rangle_{\mathsf{ct}} \overset{\blacktriangleright}{\mapsto} \langle \Omega :: [], 2\rangle_{\mathsf{ct}} \overset{\blacktriangledown}{\mapsto} \langle \Omega, [], 2\rangle_{\mathsf{ev}}$, and then the machine loops without generating any flag. Third, we skip both terms with $\langle 0 :: \Omega :: [], 2\rangle_{\mathsf{ct}} \overset{\blacktriangleright}{\mapsto} \langle \Omega :: [], 2\rangle_{\mathsf{ct}} \overset{\blacktriangleright}{\mapsto} \langle [], 2\rangle_{\mathsf{ct}} \overset{\star}{\mapsto}$. Note that the three options generate different traces of flags.

Because the rules Grab and Push are the same between the KAM and the NFB machine, there is a direct correspondence between the two.

Lemma 4.3. *For all t, t', π, π', n, $\langle t, \pi\rangle \mapsto \langle t', \pi'\rangle$ iff $\langle t, \pi, n\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto} \langle t', \pi', n\rangle_{\mathsf{ev}}$.*

We finally show that a notion of bisimilarity between configurations of an NFB machine captures normal-form bisimilarity. To this end, we first define machine bisimilarity, where we denote the flags of the machine by F.

Definition 4.4. A symmetric relation $\mathcal{R}$ is a machine bisimulation if $C_1\ \mathcal{R}\ C_2$ implies:

- if $C_1 \overset{\tau}{\mapsto}{}^{*} \overset{F}{\mapsto} C_1'$, then there exists C_2' such that $C_2 \overset{\tau}{\mapsto}{}^{*} \overset{F}{\mapsto} C_2'$ and $C_1' \,\mathcal{R}\, C_2'$;
- if $C_1 \overset{\tau}{\mapsto}{}^{*} \overset{F}{\mapsto}$, then $C_2 \overset{\tau}{\mapsto}{}^{*} \overset{F}{\mapsto}$.

Machine bisimilarity $\approx_{\mathsf{m}}$ is the largest machine bisimulation.

Intuitively, machine bisimilarity ensures that every flag emitted by a machine is matched by an identical flag from the other machine, up to internal reductions. Note that a machine that diverges with τ labels can be related to any other diverging machine or any machine stuck without a flag. We make sure the latter case cannot occur in our machines by having only terminating transitions, which are flagged, as stuck transitions. We can now state that normal-form bisimilarity coincides with machine bisimilarity of NFB machines.

Theorem 4.5. *$t \approx_{\mathsf{nf}} s$ iff there exists $n > \max(\mathsf{fv}(t) \cup \mathsf{fv}(s))$ such that $\langle t, [], n\rangle_{\mathsf{ev}} \approx_{\mathsf{m}} \langle s, [], n\rangle_{\mathsf{ev}}$.*

Proof. To prove that machine bisimilarity implies normal-form bisimilarity, we show that

$$\mathsf{R} \triangleq \{(t,s) \mid \langle t, [], n\rangle_{\mathsf{ev}} \approx_{\mathsf{m}} \langle s, [], n\rangle_{\mathsf{ev}}, n > \max(\mathsf{fv}(t) \cup \mathsf{fv}(s))\}$$

is a normal-form bisimulation. Suppose $\langle t, []\rangle \mapsto^{*} \langle \lambda x.t', []\rangle$. By Lemma 4.3, we have $\langle t, [], n\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto}{}^{*} \langle \lambda x.t', [], n\rangle_{\mathsf{ev}}$, which can only be followed by the transition $\langle \lambda x.t', [], n\rangle_{\mathsf{ev}} \overset{\circledcirc}{\mapsto} \langle [n \,/\, x]t', [], n+1\rangle_{\mathsf{ev}}$. Because $\langle t, [], n\rangle_{\mathsf{ev}} \approx_{\mathsf{m}} \langle s, [], n\rangle_{\mathsf{ev}}$, there exists a configuration C' such that $\langle s, [], n\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto}{}^{*} \overset{\circledcirc}{\mapsto} C'$ and $\langle [n/x]t', [], n+1\rangle_{\mathsf{ev}} \approx_{\mathsf{m}} C'$. The τ-steps cannot change the value of n, and the flag $\circledcirc$ ensures that there exists s' such that $\langle s, [], n\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto}{}^{*} \langle \lambda x.s', [], n\rangle_{\mathsf{ev}} \overset{\circledcirc}{\mapsto} C'$ with $C' = \langle [n \,/\, x]s', [], n+1\rangle_{\mathsf{ev}}$. Using Lemma 4.3 again, we deduce $\langle s, []\rangle \mapsto^{*} \langle \lambda x.s', []\rangle$ with $\langle [n \,/\, x]t', [], n+1\rangle_{\mathsf{ev}} \,\mathcal{R}\, \langle [n \,/\, x]s', [], n+1\rangle_{\mathsf{ev}}$, as wished. The case $\langle t, []\rangle \mapsto^{*} \langle f, \pi\rangle$ is similar.

For the reverse implication, we show that

$$\begin{aligned}\mathcal{R} \triangleq \{&(\langle t, [], n\rangle_{\mathsf{ev}}, \langle s, [], n\rangle_{\mathsf{ev}}) \mid t \approx_{\mathsf{nf}} s, n > \max(\mathsf{fv}(t) \cup \mathsf{fv}(s))\}\\ \cup \{&(\langle \pi, n\rangle_{\mathsf{ct}}, \langle \pi', n\rangle_{\mathsf{ct}}) \mid \pi \approx_{\mathsf{nf}} \pi', n > \max(\mathsf{fv}(\pi) \cup \mathsf{fv}(\pi'))\}\end{aligned}$$

is a machine bisimulation. Suppose $\langle t, [], n\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto}{}^{*} \overset{\circledcirc}{\mapsto} \langle [n \,/\, x]t', [], n+1\rangle_{\mathsf{ev}}$. By Lemma 4.3, we have $\langle t, []\rangle \mapsto^{*} \langle \lambda x.t', []\rangle$. Because $t \approx_{\mathsf{nf}} s$, there exists s' such that $\langle s, []\rangle \mapsto^{*} \langle \lambda x.s', []\rangle$ and $[n \,/\, x]t' \approx_{\mathsf{nf}} [n \,/\, x]s'$. From Lemma 4.3, we deduce $\langle s, [], n\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto}{}^{*} \overset{\circledcirc}{\mapsto} \langle [n \,/\, x]s', [], n+1\rangle_{\mathsf{ev}}$, with $\langle [n \,/\, x]t', [], n+1\rangle_{\mathsf{ev}} \,\mathcal{R}\, \langle [n \,/\, x]s', [], n+1\rangle_{\mathsf{ev}}$, as wished. The other cases are similar. □

4.3. **Translation into HOcore.** In Figure 3, we present the translation of the NFB machine into HOcore, where we consider flags as channel names. Configurations now contain a counter n, which is represented by a message on k containing the value of n encoded as a process. We use $[\![.]\!]_{\mathsf{Int}}$ to translate a natural number n into a process $\underbrace{suc(\_).\ldots.suc(\_)}_{n \text{ times}}.z(\_).\overline{init}\langle \mathbf{0}\rangle$; the role of the final output on *init* is explained later. Since we cannot generate new names in HOcore, free variables f are also numbers, and we use the same translation for them. We also use non-deterministic internal choice, encoded as follows: $P + Q \triangleq \overline{ch}\langle P\rangle \parallel \overline{ch}\langle Q\rangle \parallel ch(x).ch(\_).x$: both messages are consumed, and only one process is executed. This encoding supposes that at most one choice is active at a given time, as we use only one name *ch* to encode all the choices. We also use n-ary choices for $n > 2$ in Section 5.3, which can be encoded in the same way.

$$
\begin{aligned}
[\![t\ s]\!] &\triangleq c(p).\big([\![t]\!] \parallel \overline{c}\langle \overline{hd_{\mathsf{c}}}\langle [\![s]\!]\rangle \parallel \overline{c}\langle p\rangle\rangle\big)\\
[\![\lambda x.t]\!] &\triangleq c(p).(p \parallel \overline{b}\langle \mathsf{Restart}\rangle \parallel hd_{\mathsf{c}}(x).b(\_).[\![t]\!])\\
[\![x]\!] &\triangleq x\\
[\![f]\!] &\triangleq [\![f]\!]_{\mathsf{Int}}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Restart} &\triangleq \circledcirc(\_).k(x).(\overline{hd_{\mathsf{c}}}\langle x\rangle \parallel \overline{k}\langle suc(\_).x\rangle \parallel \overline{c}\langle [\![[]]\!]\rangle \parallel \overline{b}\langle \mathbf{0}\rangle)\\
\mathsf{Rec} &\triangleq init(\_).rec(x).(x \parallel \overline{rec}\langle x\rangle \parallel \mathsf{Cont})\\
\mathsf{Cont} &\triangleq c(p).(p \parallel \overline{b}\langle \bigstar(\_).\mathbf{0}\rangle \parallel hd_{\mathsf{c}}(x).b(\_).\mathsf{Chce}(x))\\
\mathsf{Chce}(P_t) &\triangleq \blacktriangledown(\_).c(\_).(P_t \parallel \overline{c}\langle [\![[]]\!]\rangle) + \blacktriangleright(\_).\overline{init}\langle \mathbf{0}\rangle
\end{aligned}
$$

$$
\begin{aligned}
[\![[]]\!] &\triangleq b(x).x\\
[\![t :: \pi]\!] &\triangleq \overline{hd_{\mathsf{c}}}\langle [\![t]\!]\rangle \parallel \overline{c}\langle [\![\pi]\!]\rangle\\
[\![0]\!]_{\mathsf{Int}} &\triangleq z(\_).\overline{init}\langle \mathbf{0}\rangle\\
[\![n+1]\!]_{\mathsf{Int}} &\triangleq suc(\_).[\![n]\!]_{\mathsf{Int}}
\end{aligned}
$$

$$
\begin{aligned}
[\![\langle t, \pi, n\rangle_{\mathsf{ev}}]\!] &\triangleq [\![t]\!] \parallel \overline{c}\langle [\![\pi]\!]\rangle \parallel \overline{k}\langle [\![n]\!]_{\mathsf{Int}}\rangle \parallel \mathsf{Rec} \parallel \overline{rec}\langle \mathsf{Rec}\rangle\\
[\![\langle \pi, n\rangle_{\mathsf{ct}}]\!] &\triangleq \overline{c}\langle [\![\pi]\!]\rangle \parallel \overline{k}\langle [\![n]\!]_{\mathsf{Int}}\rangle \parallel \mathsf{Cont} \parallel \mathsf{Rec} \parallel \overline{rec}\langle \mathsf{Rec}\rangle
\end{aligned}
$$

Figure 3: Translation of the NFB machine into HOcore

A stack is represented as in the KAM, by messages on hd_{c} and c, and the translation of an application $[\![t\ s]\!]$ is exactly the same as for the KAM. The encoding of the empty context [] is different, however, because contexts are used to distinguish between execution paths at two points in the machine: when evaluating a function $\lambda x.t$ in evaluation mode, and when deciding whether the execution is finished in continuation mode. The empty context is thus encoded as $b(x).x$, waiting to receive the process to execute in the empty case. For the non-empty case, this input on b is absent and there are instead messages on hd_{c} and c. Thus the generic way to choose a branch is as follows:

$$\overline{b}\langle \text{do this if empty}\rangle \parallel hd_{\mathsf{c}}(x).c(y).b(\_).\text{do this if non-empty.}$$

In the non-empty case, the input on b discards the message for the empty behavior that was not used.

For λ-abstractions, the behavior for the empty case is described in the process Restart. More precisely, $[\![\lambda x.t]\!]$ receives the current stack $[\![\pi]\!]$ on c to run it in parallel with $\overline{b}\langle \mathsf{Restart}\rangle \parallel hd_{\mathsf{c}}(x).b(\_).[\![t]\!]$. If $[\![\pi]\!]$ is of the form $\overline{hd_{\mathsf{c}}}\langle [\![t']\!]\rangle \parallel \overline{c}\langle [\![\pi']\!]\rangle$, then we have the same behavior as with the KAM, with an extra communication on b to garbage collect the Restart process.

Otherwise, $[\![\pi]\!] = b(x).x$ and we obtain the following sequence of transitions.

$$\begin{aligned}
&b(x).x \parallel \overline{b}\langle \mathsf{Restart}\rangle \parallel hd_{\mathsf{c}}(x).b(_).[\![t]\!] \parallel \overline{k}\langle [\![n]\!]_{\mathsf{Int}}\rangle \\
\xrightarrow{\tau}\ &\circledcirc\,(_).k(x).(\overline{hd_{\mathsf{c}}}\langle x\rangle \parallel \overline{k}\langle suc(_).x\rangle \parallel \overline{c}\langle [\![[]]\!]\rangle \parallel \overline{b}\langle \mathbf{0}\rangle) \parallel hd_{\mathsf{c}}(x).b(_).[\![t]\!] \parallel \overline{k}\langle [\![n]\!]_{\mathsf{Int}}\rangle \\
\xRightarrow{\circledcirc}\ &\overline{hd_{\mathsf{c}}}\langle [\![n]\!]_{\mathsf{Int}}\rangle \parallel \overline{k}\langle suc(_).[\![n]\!]_{\mathsf{Int}}\rangle \parallel \overline{c}\langle [\![[]]\!]\rangle \parallel \overline{b}\langle \mathbf{0}\rangle \parallel hd_{\mathsf{c}}(x).b(_).[\![t]\!] \\
\xrightarrow{\tau}\ &\overline{k}\langle suc(_).[\![n]\!]_{\mathsf{Int}}\rangle \parallel \overline{c}\langle [\![[]]\!]\rangle \parallel \overline{b}\langle \mathbf{0}\rangle \parallel b(_).[\![[n\,/\,x]t]\!] \\
\xrightarrow{\tau}\ &\overline{k}\langle [\![n+1]\!]_{\mathsf{Int}}\rangle \parallel \overline{c}\langle [\![[]]\!]\rangle \parallel [\![[n\,/\,x]t]\!]
\end{aligned}$$

In the end, we have effectively restarted the machine to evaluate $[n\,/\,x]t$, as wished.

In continuation mode, the branching is done by the process Cont, which is executed after applying the transition Var. More precisely, a free variable f is translated using $[\![.]\!]_{\mathsf{Int}}$, which signals first the value of f (with the names *suc* and *z*), and then sends a message on *init* to enter the continuation mode. The way the NFB machine chooses which t_i to evaluate in a stack $t_1 :: \ldots :: t_m ::[]$ is a recursive mechanism, and recursion can be encoded in a higher-order calculus: $\mathsf{Rec} \parallel \overline{rec}\langle \mathsf{Rec}\rangle$ reduces to $\mathsf{Cont} \parallel \mathsf{Rec} \parallel \overline{rec}\langle \mathsf{Rec}\rangle$ when it receives a message on *init*. The process Cont is doing a case analysis on $[\![\pi]\!]$ when it is executed in parallel with $\overline{c}\langle [\![\pi]\!]\rangle$: if $\pi = []$, then $[\![\pi]\!] = b(x).x$ receives the message on b which flags ☆ and the machine stops. Otherwise, $[\![\pi]\!] = \overline{hd_{\mathsf{c}}}\langle [\![t]\!]\rangle \parallel \overline{c}\langle [\![\pi']\!]\rangle$, and we have the following reductions:

$$\begin{aligned}
\mathsf{Cont} \parallel \overline{c}\langle [\![\pi]\!]\rangle &\xrightarrow{\tau} \overline{hd_{\mathsf{c}}}\langle [\![t]\!]\rangle \parallel \overline{c}\langle [\![\pi']\!]\rangle \parallel \overline{b}\langle \star(_).\mathbf{0}\rangle \parallel hd_{\mathsf{c}}(x).b(_).\mathsf{Chce}(x) \\
&\xrightarrow{\tau}{}^{2}\ \overline{c}\langle [\![\pi']\!]\rangle \parallel \mathsf{Chce}([\![t]\!])
\end{aligned}$$

At this point, $\mathsf{Chce}([\![t]\!])$ either evaluates t with flag ▼, or flags ▶ and continues exploring π'. In the former case, the current stack π' is replaced by an empty stack, and in the latter, a message on *init* is issued to produce $\mathsf{Cont} \parallel \overline{c}\langle [\![\pi']\!]\rangle$ after some reduction steps.

4.4. **Operational Correspondence and Full Abstraction.** Establishing full abstraction requires first to state the correspondence between the NFB machine and its translation. We call *machine process* a process obtained by reducing a translated configuration.

Definition 4.6. A process P is a machine process if there exists a configuration C of the machine such that $[\![C]\!] \xRightarrow{\tau} P$.

We recall that F range over the flags of the NFB machine, i.e., ⊚, ▼, ▶, ☆, and f. We let $\hat{F}$ range over flags and τ. The translation maps the flags to corresponding channel names, except for f, which is represented using the *suc* and *z* channel names. We call *flag names* the channel names ⊚, ▼, ▶, ☆, *suc*, and *z*. We write $\xrightarrow{f}$ as a shorthand for f transitions $\xrightarrow{suc}$ followed by a transition $\xrightarrow{z}$, i.e., $(\xrightarrow{suc})^f \xrightarrow{z}$, and $\xRightarrow{f}$ stands for $\xRightarrow{\tau}\xrightarrow{f}\xRightarrow{\tau}$. We define $\mathbb{H}$ as the set of names in the translation that are not flag names. We write $\downarrow_\gamma$ as a shorthand for $\downarrow_\gamma^{\mathbb{H}}$, $\Downarrow_\gamma$ for $\Downarrow_\gamma^{\mathbb{H}}$, $\mathsf{WkObs}(P)$ for $\mathsf{WkObs}_{\mathbb{H}}(P)$, and $\approx_{\mathsf{HO}}$ for $\approx_{\mathsf{HO}}^{\mathbb{H}}$.

Given a machine process P and a flag $\hat{F}$, we define a set $\mathsf{next}(\hat{F}, P)$ such that $Q \in \mathsf{next}(\hat{F}, P)$ implies there is a weak reduction between P and Q with at least one transition labeled $\hat{F}$, Q is the translation of a configuration, and there is no other machine translation

in between. Intuitively, it is the set of translations that are reachable from P just after a step $\hat{F}$.

Definition 4.7. We define $\mathsf{next}(\hat{F}, P)$ so that $Q \in \mathsf{next}(\hat{F}, P)$ implies $P \stackrel{\tau}{\Longrightarrow} \xrightarrow{\hat{F}} \stackrel{\tau}{\Longrightarrow} Q$, $Q = [\![C']\!]$ for some C', and for any P' such that $P' \neq P$, $P' \neq Q$, and $P \stackrel{\hat{F}}{\Longrightarrow} P' \stackrel{\tau}{\Longrightarrow} Q$ or $P \stackrel{\tau}{\Longrightarrow} P' \stackrel{\hat{F}}{\Longrightarrow} Q$, we have $P' \neq [\![C]\!]$ for any C.

The translation has been designed so that each machine process is either stuck waiting for a message on a flag name, or has at most one possible communication, except when a non-deterministic choice has to be made. The latter case occurs when the NFB machine is exploring a stack, with a translated configuration of the form $[\![\langle t :: \pi, n\rangle_{\mathsf{ct}}]\!]$: as explained at the end of Section 4.3, we have

$$[\![\langle t :: \pi, n\rangle_{\mathsf{ct}}]\!] \stackrel{\tau}{\Longrightarrow} \mathsf{Chce}([\![t]\!]) \parallel \overline{c}\langle[\![\pi]\!]\rangle \parallel \overline{k}\langle[\![n]\!]\rangle \parallel \mathsf{Rec} \parallel \overline{rec}\langle\mathsf{Rec}\rangle$$

We call the resulting process $\mathsf{Choice}(t, \pi, n) \triangleq \mathsf{Chce}([\![t]\!]) \parallel \overline{c}\langle[\![\pi]\!]\rangle \parallel \overline{k}\langle[\![n]\!]\rangle \parallel \mathsf{Rec} \parallel \overline{rec}\langle\mathsf{Rec}\rangle$ a *choice process*, as it is about to make an arbitrary choice in $\mathsf{Chce}([\![t]\!])$. Unfolding the encoding of choice, we have $\mathsf{Chce}([\![t]\!]) = \overline{ch}\langle P_{\blacktriangledown,t}\rangle \parallel \overline{ch}\langle P_{\blacktriangleright}\rangle \parallel ch(x).ch(\_).x$, with $P_{\blacktriangledown,t} \triangleq \blacktriangledown(\_).c(\_).([\![t]\!] \parallel \overline{c}\langle[\![[]]\!]\rangle)$ and $P_{\blacktriangleright} \triangleq \blacktriangleright(\_).\overline{init}\langle\mathbf{0}\rangle$. Let $P_{\pi,n} \triangleq \overline{c}\langle[\![\pi]\!]\rangle \parallel \overline{k}\langle[\![n]\!]\rangle \parallel \mathsf{Rec} \parallel \overline{rec}\langle\mathsf{Rec}\rangle$. Then the two possible transitions from $\mathsf{Choice}(t, \pi, n)$ lead to the following machine translations:

$$\mathsf{Choice}(t,\pi,n) \xrightarrow{\tau} \overline{ch}\langle P_{\blacktriangleright}\rangle \parallel ch(\_).P_{\blacktriangledown,t} \parallel P_{\pi,n} \xrightarrow{\tau} P_{\blacktriangledown,t} \parallel P_{\pi,n} \xrightarrow{\blacktriangledown} \xrightarrow{\tau}{}^{2} [\![\langle t, [], n\rangle_{\mathsf{ev}}]\!]$$

$$\mathsf{Choice}(t,\pi,n) \xrightarrow{\tau} \overline{ch}\langle P_{\blacktriangledown,t}\rangle \parallel ch(\_).P_{\blacktriangleright} \parallel P_{\pi,n} \xrightarrow{\tau} P_{\blacktriangleright} \parallel P_{\pi,n} \xrightarrow{\blacktriangleright} \xrightarrow{\tau}{}^{2} [\![\langle \pi, n\rangle_{\mathsf{ct}}]\!]$$

The first transition $\xrightarrow{\tau}$ is making the choice. From there, one process flags $\blacktriangledown$, while the other flags $\blacktriangleright$: we know which choice has been made by looking at the weak observable actions. Note that $\mathsf{next}(\tau, \mathsf{Choice}(t, \pi, n))$ is empty, because we need to flag either $\blacktriangledown$ or $\blacktriangleright$ to reach a machine translation.

The next lemma expresses the fact that the translation is deterministic, relying on flags in the case of choice processes.

Lemma 4.8 (Determinism)**.** *Let P be a machine process.*

- *If P is not a choice process, then either P cannot reduce, or there exist a unique P' and $\hat{F}$ such that $P \xrightarrow{\hat{F}} P'$;*
- *if P is a choice process, then there exist P_1 and P_2 such that $P \xrightarrow{\tau} P_1$, $P \xrightarrow{\tau} P_2$, and $\mathsf{WkObs}(P_1) \neq \mathsf{WkObs}(P_2)$. For all $\hat{F}$, P' such that $P \xrightarrow{\hat{F}} P'$, we have $\hat{F} = \tau$ and either $P' = P_1$ or $P' = P_2$.*

Proof. The proof is a straightforward case analysis on the shape of the machine processes, checking that for each of them, the lemma holds. □

Lemma 4.8 implies that $\mathsf{next}(\hat{F}, P)$ is either empty or a singleton for every $\hat{F}$ and P. In the following, we write $\mathsf{next}(\hat{F}, P)$ to assert that it is not empty and to directly denote the corresponding unique machine translation.

A consequence is that a machine process may only have flag names as possible observable actions. In particular, it cannot have a coname $\overline{a}$ as an observable action.

Corollary 4.9. *For all machine process P, $\mathsf{WkObs}(P) \subseteq \{\circledcirc, \blacktriangledown, \blacktriangleright, \bigstar, suc, z\}$.*

The second property we need is an operational correspondence between the machine and its translation.

Lemma 4.10 (Faithfulness)**.** *The following holds:*

- $C \overset{\hat{F}}{\mapsto} C'$ *iff* $\mathsf{next}(\hat{F}, [\![C]\!]) = [\![C']\!]$*;*
- $C \overset{\bigstar}{\mapsto}$ *iff* $[\![C]\!] \overset{\tau}{\Longrightarrow} \overset{\bigstar}{\longrightarrow} P$ *and* $P \approx_{\mathsf{HO}} \mathbf{0}$*.*

Proof. The proof is by a straightforward case analysis on C in the "only if" direction, and by case analysis on $[\![C]\!]$ in the "if" direction. □

With these properties, we can prove that the translation from NFB machines to HOcore processes is fully abstract. We define the process complementing a flag as follows: for a flag $F \neq f$, we write $\overline{F}$ for the process $\overline{F}\langle \mathbf{0} \rangle$. We define $\overline{f}$ inductively so that $\overline{n+1} \triangleq \overline{suc}\langle \mathbf{0} \rangle \parallel \overline{n}$ and $\overline{0} \triangleq \overline{z}\langle \mathbf{0} \rangle$. The following lemma allows us to reason up to τ-transitions on the HOcore side.

Lemma 4.11. *Let P, P' be machine processes. If $P \overset{\tau}{\Longrightarrow} P'$ and $\mathsf{WkObs}(P) = \mathsf{WkObs}(P')$, then $P \approx_{\mathsf{HO}} P'$.*

Proof. One can check that $\{(P \parallel R, P' \parallel R) \mid P \overset{\tau}{\Longrightarrow} P', \mathsf{WkObs}(P) = \mathsf{WkObs}(P')\} \cup \{(R, R)\}$ is a barbed bisimulation using Lemma 4.8. □

Lemma 4.12. *If $[\![C]\!] \approx_{\mathsf{HO}} [\![C']\!]$, then $C \approx_{\mathsf{m}} C'$.*

Proof. We prove that $\mathsf{R} \triangleq \{(C, C') \mid [\![C]\!] \approx_{\mathsf{HO}} [\![C']\!]\}$ is a machine bisimulation. Let $C_1 \mathcal{R} C_1'$.

Suppose $C_1 \overset{\tau}{\mapsto}{}^{*} C_2 \overset{F}{\mapsto} C_3$. By Lemma 4.10, we have

$$[\![C_1]\!] \parallel \overline{F} \overset{\tau}{\Longrightarrow} [\![C_2]\!] \parallel \overline{F} \overset{\tau}{\Longrightarrow} [\![C_3]\!].$$

Because $[\![C_1]\!] \approx_{\mathsf{HO}} [\![C_1']\!]$, there exists P such that $[\![C_1']\!] \parallel \overline{F} \overset{\tau}{\Longrightarrow} P$ and $[\![C_3]\!] \approx_{\mathsf{HO}} P$. Since $\mathsf{WkObs}([\![C_3]\!]) = \mathsf{WkObs}(P)$, we have in particular $\neg(P \downarrow_{\overline{F}})$ by Corollary 4.9. Consequently, there exists P' such that $[\![C_1']\!] \overset{\tau}{\Longrightarrow} P'$, $P' \downarrow_F$, and $P' \parallel \overline{F} \overset{\tau}{\Longrightarrow} P$. Consider $\mathsf{next}(F, P') = [\![C_3']\!]$ for some C_3', which must exists since P' still has a barb on F. We have $[\![C_1']\!] \overset{\tau}{\Longrightarrow} P' \overset{F}{\Longrightarrow} [\![C_3']\!]$, which implies $C_1' \overset{\tau}{\mapsto}{}^{*} \overset{F}{\mapsto} C_3'$ by Lemma 4.10. To show that $C_3 \mathcal{R} C_3'$, we need to prove that $[\![C_3]\!] \approx_{\mathsf{HO}} [\![C_3']\!]$. Because $[\![C_3]\!] \approx_{\mathsf{HO}} P$, it is enough to prove $P \approx_{\mathsf{HO}} [\![C_3']\!]$.

Suppose there is a choice process Q so that $P' \parallel \overline{F} \overset{\tau}{\Longrightarrow} Q \overset{\tau}{\Longrightarrow} [\![C_3']\!]$. It means that a communication on $F' \in \{\blacktriangledown, \blacktriangleright\}$ happens, as it is the only possibility for a choice process to reach a translated configuration. The communication cannot be with $\overline{F}$, because that output is consumed by P'. Therefore, it implies that $P' \Downarrow_{\overline{F'}}$, but P' is a machine process, so it is not possible by Corollary 4.9.

In the end, we have both $P' \parallel \overline{F} \overset{\tau}{\Longrightarrow} P$ and $P' \parallel \overline{F} \overset{\tau}{\Longrightarrow} [\![C_3']\!]$, and there is no choice process between $P' \parallel \overline{F}$ and $[\![C_3']\!]$. As this is the only source of non-determinism, by Lemma 4.8, we either have $P \overset{\tau}{\Longrightarrow} [\![C_3']\!]$ or $[\![C_3']\!] \overset{\tau}{\Longrightarrow} P$.

We now prove that $\mathsf{WkObs}(\llbracket C_3' \rrbracket) = \mathsf{WkObs}(P)$, to be able to conclude with Lemma 4.11. The only way to have $\mathsf{WkObs}(\llbracket C_3' \rrbracket) \neq \mathsf{WkObs}(P)$ is if a choice has been made between $\llbracket C_3' \rrbracket$ and P, i.e., if there is a choice process Q making at least one τ-transition between the two. The case $P \overset{\tau}{\Longrightarrow} Q \overset{\tau}{\longrightarrow\!\!\!\!\Longrightarrow} \llbracket C_3' \rrbracket$ has been ruled out, because there is no choice process between $P' \parallel \overline{F}$ and $\llbracket C_3' \rrbracket$. The other possibility is $\llbracket C_3' \rrbracket \overset{\tau}{\Longrightarrow} Q \overset{\tau}{\longrightarrow\!\!\!\!\Longrightarrow} P$, where Q is a choice process. Because a choice has been made, we have either $\mathsf{WkObs}(P) \subseteq \{\blacktriangledown\}$ or $\mathsf{WkObs}(P) \subseteq \{\blacktriangleright\}$. However, we have $\mathsf{WkObs}(P) = \mathsf{WkObs}(\llbracket C_3 \rrbracket)$, and the translation of a configuration cannot have only a single flag of a non-deterministic choice as weak observable action, which leads to a contradiction. Consequently, we have $\mathsf{WkObs}(P) = \mathsf{WkObs}(\llbracket C_3' \rrbracket)$, hence $P \approx_{\mathsf{HO}} \llbracket C_3' \rrbracket$ by Lemma 4.11.

Suppose $C_1 \overset{\tau}{\mapsto}{}^{*}\ C_2 \overset{\bigstar}{\mapsto}$; then $\llbracket C_1 \rrbracket \parallel \overline{\bigstar} \overset{\tau}{\Longrightarrow} \llbracket C_2 \rrbracket \parallel \overline{\bigstar} \overset{\tau}{\Longrightarrow} P$ with $P \approx_{\mathsf{HO}} \mathbf{0}$ by Lemma 4.10. Therefore, there exists P' such that $\llbracket C_1' \rrbracket \parallel \overline{\bigstar} \overset{\tau}{\Longrightarrow} P'$ and $P \approx_{\mathsf{HO}} P' \approx_{\mathsf{HO}} \mathbf{0}$. With Lemma 4.10, we have $C_1' \overset{\tau}{\mapsto}{}^{*} \overset{\bigstar}{\mapsto}$, as wished. □

Lemma 4.13. *If $C \approx_{\mathsf{m}} C'$, then $\llbracket C \rrbracket \approx_{\mathsf{HO}} \llbracket C' \rrbracket$.*

Proof. We prove that

$$\mathcal{R} \triangleq \left\{ \begin{pmatrix} (P \parallel R, Q \parallel R) \mid C \approx_{\mathsf{m}} C', \mathsf{WkObs}(P) = \mathsf{WkObs}(Q), \mathsf{fn}(R) \cap \mathbb{H} = \varnothing \\ P \overset{\tau}{\Longrightarrow} \llbracket C \rrbracket, Q \overset{\tau}{\Longrightarrow} \llbracket C' \rrbracket \text{ or } \llbracket C \rrbracket \overset{\tau}{\Longrightarrow} P, \llbracket C' \rrbracket \overset{\tau}{\Longrightarrow} Q \end{pmatrix} \right\}$$
$$\cup \{(P \parallel R, Q \parallel R) \mid P \approx_{\mathsf{HO}} \mathbf{0} \approx_{\mathsf{HO}} Q, \mathsf{fn}(R) \cap \mathbb{H} = \varnothing\}$$

is a barbed bisimulation. The congruence condition holds by definition of $\mathcal{R}$, and the observable actions are the same because of the condition $\mathsf{WkObs}(P) = \mathsf{WkObs}(Q)$. The reductions from R are easy to match; we thus only check the reductions involving P or Q.

If $P \overset{\tau}{\longrightarrow} P'$, then we show that there exists Q' such that $Q \parallel R \overset{\tau}{\Longrightarrow} Q' \parallel R$ and $P' \parallel R \mathrel{\mathcal{R}} Q' \parallel R$. If $P \overset{\tau}{\Longrightarrow} \llbracket C \rrbracket$ and $Q \overset{\tau}{\Longrightarrow} \llbracket C' \rrbracket$, then taking $Q' = \llbracket C' \rrbracket$ works. If $\llbracket C \rrbracket \overset{\tau}{\Longrightarrow} P$ and $\llbracket C' \rrbracket \overset{\tau}{\Longrightarrow} Q$, then we distinguish two cases. If P is not a choice process, then $\mathsf{WkObs}(P) = \mathsf{WkObs}(P')$, and we can simply choose $Q' = Q$. Otherwise, the transition $P \overset{\tau}{\longrightarrow} P'$ is making a choice and $\mathsf{WkObs}(P') \subsetneq \mathsf{WkObs}(P)$. But $\mathsf{WkObs}(P) = \mathsf{WkObs}(Q)$ implies that Q also reduces to a choice process. Then it is possible to choose the same branch as P' and to define Q' accordingly.

Suppose $P \parallel R \overset{\tau}{\longrightarrow} P'$ with a communication on a flag F—a communication on another name is not possible because $\mathsf{fn}(R) \cap \mathbb{H} = \varnothing$. Then $R \downarrow_{\overline{F}}$ and $P \downarrow_F$, which is possible only in the case $\llbracket C \rrbracket \overset{\tau}{\Longrightarrow} P$. Therefore we also have $\llbracket C' \rrbracket \overset{\tau}{\Longrightarrow} Q$ by definition of $\mathcal{R}$.

Suppose $F \neq \bigstar$: then we also have $P \overset{F}{\longrightarrow} P'' \overset{\tau}{\Longrightarrow} \llbracket C'' \rrbracket$ for some P'' with $\llbracket C'' \rrbracket = \mathsf{next}(F, P)$. By Lemma 4.8, there exist R' such that $P' = P'' \parallel R'$. We have $\llbracket C \rrbracket \overset{F}{\Longrightarrow} \llbracket C'' \rrbracket$, which implies $C \overset{\tau}{\mapsto}{}^{*} \overset{F}{\mapsto} C''$ by Lemma 4.10. Because $C \approx_{\mathsf{m}} C'$, there exists C''' such that $C' \overset{\tau}{\mapsto}{}^{*} \overset{F}{\mapsto} C'''$ and $C'' \approx_{\mathsf{m}} C'''$. By Lemma 4.10, this implies $\llbracket C' \rrbracket \overset{F}{\Longrightarrow} \llbracket C''' \rrbracket$; in particular,

there exists Q'' such that $[\![C']\!] \xRightarrow{\tau} \xrightarrow{F} Q'' \xRightarrow{\tau} [\![C'''']\!]$. By Lemma 4.8, we also have $Q \xRightarrow{\tau} \xrightarrow{F} Q''$, therefore $Q \parallel R \xRightarrow{\tau} Q'' \parallel R'$. To show that $P'' \parallel R' \mathrel{\mathcal{R}} Q'' \parallel R'$, what is left to check is $\mathsf{WkObs}(P'') = \mathsf{WkObs}(Q'')$. We have $P'' \xRightarrow{\tau} [\![C'']\!]$, so there is no choice process between P'' and $[\![C'']\!]$, because a transition with label $\blacktriangledown$ or $\blacktriangleright$ is necessary to go from a choice process to the translation of a configuration. Therefore, we have $\mathsf{WkObs}(P'') = \mathsf{WkObs}([\![C'']\!])$, and similarly $\mathsf{WkObs}(Q'') = \mathsf{WkObs}([\![C''']\!])$. Because $C'' \approx_{\mathsf{m}} C'''$, any action from C'' is matched by C''' and conversely, which implies $\mathsf{WkObs}([\![C'']\!]) = \mathsf{WkObs}([\![C''']\!])$ by Lemma 4.10. In the end, we have $\mathsf{WkObs}(P'') = \mathsf{WkObs}(Q'')$, and therefore $P'' \parallel R' \mathrel{\mathcal{R}} Q'' \parallel R'$, as wished.

If $F = \bigstar$, then $P \xrightarrow{\bigstar} P''$ for some P'' such that $P'' \approx_{\mathsf{HO}} \mathbf{0}$. By Lemma 4.8, there exists R' such that $P' = P'' \parallel R'$. We have $[\![C]\!] \xRightarrow{\tau} \xrightarrow{\bigstar} P''$, so $C \overset{\tau}{\mapsto}{}^{*} \overset{\bigstar}{\mapsto}$ holds by Lemma 4.10. Because $C \approx_{\mathsf{m}} C'$, we have $C' \overset{\tau}{\mapsto}{}^{*} \overset{\bigstar}{\mapsto}$, so by Lemma 4.10, there exists Q'' such that $[\![C']\!] \xRightarrow{\tau} \xrightarrow{\bigstar} Q''$ and $Q'' \approx_{\mathsf{HO}} \mathbf{0}$. By Lemma 4.8, we also have $Q \xRightarrow{\tau} \xrightarrow{\bigstar} Q''$, therefore we have $Q \parallel R \xRightarrow{\tau} Q'' \parallel R'$. The resulting processes $P'' \parallel R'$ and $Q'' \parallel R'$ are in the second set defining $\mathcal{R}$. □

Theorem 4.14. *$C \approx_{\mathsf{m}} C'$ iff $[\![C]\!] \approx_{\mathsf{HO}} [\![C']\!]$.*

Proof. By Lemmas 4.12 and 4.13. □

As a result, we can deduce full abstraction between HOcore and the λ-calculus with normal-form bisimilarity.

Corollary 4.15. *$t \approx_{\mathsf{nf}} s$ iff there exists $n > \max(\mathsf{fv}(t) \cup \mathsf{fv}(s))$ such that $[\![\langle t, [], n\rangle_{\mathsf{ev}}]\!] \approx_{\mathsf{HO}} [\![\langle s, [], n\rangle_{\mathsf{ev}}]\!]$.*

Proof. By Theorems 4.5 and 4.14. □

The proofs of Lemmas 4.12 and 4.13 rely on the translation being deterministic (Lemma 4.8) and faithful (Lemma 4.10). In particular, flagged non-determinism in machines is represented by choice processes in HOcore, so that these processes make the choice observable with different flags before reducing to the next translated configurations. The machine translations defined in the upcoming sections satisfy these properties, and the correspondence between machines and their translation in HOcore can be proved in a similar fashion.

5. Applicative Bisimilarity

Proving full abstraction w.r.t. normal-form bisimilarity requires minimal interactions—synchronizations on flags—between a machine process and the outside. Achieving full abstraction w.r.t. applicative bisimilarity is intuitively more difficult, since this bisimilarity tests λ-abstractions by applying them to an arbitrary argument. Internalizing such bisimilarity is simple using higher-order flags: one may think of the following transition to test the result of a computation:

$$\langle \lambda x.t, [] \rangle \overset{s}{\mapsto} \langle [s \,/\, x]t, [] \rangle$$

Although HOcore has higher-order communications, we cannot use them to obtain a fully abstract encoding of such a machine for two reasons. First, allowing interactions where the environment provides a term may allow arbitrary processes to be received, including processes that are not in the image of the translation, thus potentially breaking invariants of the translation. Second, the translation of the KAM has to hide the names it uses for the translation to be fully abstract; it is thus impossible for the context to use such names and to provide translated λ-terms to be tested.

We thus propose in this section to internalize applicative bisimilarity using ordinary flags: when the abstract machine reaches a value, it switches to a different mode where it non-deterministically builds a test term step by step, using flags to indicate its choices so as to ensure that a bisimilar machine builds the same term. The translation of such a machine into HOcore is then similar to the translation of the NFB machine.

Using simple flags to generate terms step by step implies we need to deal with binders. In particular, and anticipating on the HOcore translation, we no longer can rely on the definition of binding and substitution from HOcore, as we cannot write a process that inputs a translation of t and outputs a translation of $\lambda x.t$ using an HOcore binding for x. We thus switch to a pure data description of bindings, using de Bruijn indices. As such terms still need to be executed, we first recall the definition of the KAM with de Bruijn indices and the definitions of contextual equivalence and applicative bisimilarity for λ-terms with de Bruijn indices. We then present the machine internalizing applicative bisimilarity, its translation into HOcore, and show they are fully abstract. We finally conclude this section by showing how contextual equivalence is internalized in an abstract machine, generating contexts instead of terms.

5.1. **The KAM and Behavioral Equivalences.** In the λ-calculus with de Bruijn indices, a variable is a number that indicates how many λ's are between the variable and its binder. For example, $\lambda x.x$ is written $\lambda.0$ and $\lambda xy.x\ y$ is written $\lambda.\lambda.1\ 0$. The syntax of terms (t, s), closures (η), environments (e, d), and values (v) is as follows.

$$t, s ::= n \mid t\ s \mid \lambda.t \qquad \eta ::= (t, e) \qquad e, d ::= \eta :: e \mid \epsilon \qquad v ::= (\lambda.t, e)$$

A *closure* η is a pair (t, e) where e is an environment mapping the free variables of t to closures; environments are used in lieu of substitutions. A term t is closed if $\mathsf{fv}(t) = \varnothing$, and a closure (t, e) is closed if the number of elements of e is bigger than the highest free variable of t, and e is composed only of closed closures.

The semantics is given by the original, environment-based KAM, where a configuration C is now composed of the closed closure (t, e) being evaluated, and a stack π of closures. The transitions rules are as follows.

$$C ::= \langle t, e, \pi\rangle_{\mathsf{ev}}\ \text{(configurations)} \qquad \pi ::= \eta :: \pi \mid [\,]\ \text{(stacks)}$$

$$\langle t\ s, e, \pi\rangle_{\mathsf{ev}} \xrightarrow{\tau} \langle t, e, (s, e) :: \pi\rangle_{\mathsf{ev}} \tag{Push}$$

$$\langle 0, (t, e) :: d, \pi\rangle_{\mathsf{ev}} \xrightarrow{\tau} \langle t, e, \pi\rangle_{\mathsf{ev}} \tag{Zero}$$

$$\langle n + 1, (t, e) :: d, \pi\rangle_{\mathsf{ev}} \xrightarrow{\tau} \langle n, d, \pi\rangle_{\mathsf{ev}} \tag{Env}$$

$$\langle \lambda.t, e, \eta :: \pi\rangle_{\mathsf{ev}} \xrightarrow{\tau} \langle t, \eta :: e, \pi\rangle_{\mathsf{ev}} \tag{Grab}$$

In Push, the argument s of an application is stored on the stack with its environment e while the term t in function position is evaluated. If we get a λ-abstraction $\lambda.t$ (transition

Grab), then an argument η is moved from the stack to the top of the environment to remember that η corresponds to the de Bruijn index 0, and the evaluation continues with t. Looking up the closure corresponding to a de Bruijn index in the environment is done with the rules Env and Zero. Because we evaluate closed closures only, it is not possible to obtain a configuration of the form $\langle n, \epsilon, \pi\rangle_{\mathsf{ev}}$. If a configuration of the form $\langle \lambda.t, e, []\rangle_{\mathsf{ev}}$ is reached, then the evaluation is finished, and the result is $(\lambda.t, e)$.

Behavioral equivalences. Contextual equivalence compares closed terms by testing them within all contexts. A context $\mathcal{C}$ is a term with a hole $\Box$ at a variable position; plugging a term t in $\mathcal{C}$ is written $\mathcal{C}[t]$. A context is closed if $\mathsf{fv}(\mathcal{C}) = \varnothing$. Contextual equivalence is then defined as follows.

Definition 5.1. Two closed terms t and s are contextually equivalent, written $t \approx_{\mathsf{ctx}} s$, if for all closed contexts $\mathcal{C}$, $\langle \mathcal{C}[t], \epsilon, []\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*} \langle \lambda.t', e, []\rangle_{\mathsf{ev}}$ for some t' and e iff $\langle \mathcal{C}[s], \epsilon, []\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*}$ $\langle \lambda.s', d, []\rangle_{\mathsf{ev}}$ for some s' and d.

Contextual equivalence is characterized by applicative bisimilarity [AO93], which reduces closed terms to values that are then applied to an arbitrary argument.

Definition 5.2. A symmetric relation $\mathcal{R}$ on closed closures is an applicative bisimulation if $(t, e)\ \mathcal{R}\ (s, d)$ and $\langle t, e, []\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*} \langle \lambda.t', e', []\rangle_{\mathsf{ev}}$ implies that there exist s' and d' such that $\langle s, d, []\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*} \langle \lambda.s', d', []\rangle_{\mathsf{ev}}$, and for all closed t'', we have $(t', (t'', \epsilon) :: e')\ \mathcal{R}\ (s', (t'', \epsilon) :: d')$.

Applicative bisimilarity $\approx_{\mathsf{app}}$ is the largest applicative bisimulation.

We can prove full abstraction between HOcore and the λ-calculus by either internalizing contextual equivalence or applicative bisimilarity. We choose the latter, as it is closer to normal-form bisimilarity. We briefly show in Section 5.4 how the machine of Section 5.2 can be adapted to handle contextual equivalence.

5.2. **Argument Generation for the Applicative Bisimilarity.** After evaluating a term thanks to the KAM, we want to produce a closed argument to pass it to the resulting value, and then restart the evaluation process. The approach of [CF11] consists in waiting for a name from the outside, giving access to a translated λ-term to be applied to. If the environment does not provide access to a well-formed translation, the process simulating β-reduction remains stuck. In contrast, our machine directly generates a well-formed argument: we represent a λ-term as a syntax tree, with de Bruijn indices at the leaves, and applications and λ-abstractions at the nodes. We start generating from the leftmost variable, and we then go left to right, meaning that in an application, we create the term in function position before the argument. These choices are completely arbitrary, as doing the opposite—starting from the rightmost variable and go right to left—is also possible. To be sure that we produce a valid, closed, λ-term, we have to check that each de Bruijn index n has at least $n + 1$ λ-abstractions enclosing it, and that each application node has two children.

To do so, we consider machine states with four components: the term t being constructed, a counter κ giving the minimal number of λ-abstractions required to close the term, a stack ρ used to build applications, whose syntax is

$$\rho ::= (t, \kappa) :: \rho \mid \odot \qquad \text{(application stacks)}$$

and which is explained in more detail later, and finally the closure η for which the argument is being built. This last element is never modified by the building process, and is just used

$$\langle \lambda.t, e, [] \rangle_{\mathsf{ev}} \overset{\circledcirc}{\longmapsto} \langle 0, \odot, (t, e) \rangle_{\mathsf{ind}} \tag{Arg}$$

$$\langle n, \rho, \eta \rangle_{\mathsf{ind}} \overset{\boxplus}{\longmapsto} \langle n+1, \rho, \eta \rangle_{\mathsf{ind}} \tag{Suc}$$

$$\langle n, \rho, \eta \rangle_{\mathsf{ind}} \overset{\boxdot}{\longmapsto} \langle n, n+1, \rho, \eta \rangle_{\mathsf{tm}} \tag{Var}$$

$$\langle t, \kappa+1, \rho, \eta \rangle_{\mathsf{tm}} \overset{\lambda}{\longmapsto} \langle \lambda.t, \kappa, \rho, \eta \rangle_{\mathsf{tm}} \tag{Lambda}$$

$$\langle t, 0, \rho, \eta \rangle_{\mathsf{tm}} \overset{\lambda}{\longmapsto} \langle \lambda.t, 0, \rho, \eta \rangle_{\mathsf{tm}} \tag{Lambda0}$$

$$\langle t, \kappa, \rho, \eta \rangle_{\mathsf{tm}} \overset{\dashv\!\shortmid}{\longmapsto} \langle 0, (t, \kappa) :: \rho, \eta \rangle_{\mathsf{ind}} \tag{AppFun}$$

$$\langle s, \kappa_1, (t, \kappa_2) :: \rho, \eta \rangle_{\mathsf{tm}} \overset{@}{\longmapsto} \langle t\ s, \mathsf{max}(\kappa_1, \kappa_2), \rho, \eta \rangle_{\mathsf{tm}} \tag{App}$$

$$\langle t, 0, \odot, (s, e) \rangle_{\mathsf{tm}} \overset{\bigstar}{\longmapsto} \langle s, (t, \epsilon) :: e, [] \rangle_{\mathsf{ev}} \tag{Restart}$$

Figure 4: AB machine: argument generation

to restart the machine in evaluation mode when the argument is finished. We distinguish two kinds of states: the index state $\langle n, \rho, \eta \rangle_{\mathsf{ind}}$, where only de Bruijn indices can be built, and the term state $\langle t, \kappa, \rho, \eta \rangle_{\mathsf{tm}}$, where any term can be produced. The transitions for these states are given in Figure 4, and we call the extension of the KAM with these transitions the AB machine.

The transition Arg starts the building process when we reach a λ-abstraction in evaluation mode with the empty continuation []. We begin with the index 0 and with the empty stack ⊙. The value of the index can then be increased with the transition Suc. When we reach the needed value for the index, the transition Var switches to the term mode, indicating that we need at least $n+1$ λ-abstractions to close the term. We use two modes to prevent a Suc transition on a term which is not an index.

In term mode, we can add λ-abstractions to the term, decreasing κ if $\kappa > 0$ with transition Lambda, or leaving κ at 0 with transition Lambda0; the abstractions we introduce when $\kappa = 0$ do not bind any variable. Once we are done building a term t in function position of an application, we use transition AppFun to build the argument s. We start again in index mode, but we store on top of ρ the term t with its counter κ_2. When we finish s with a counter κ_1, we build the application with transition App, which takes the maximum of κ_1 and κ_2 as the new minimal number of λ-abstractions needed above $t\ s$. Note that the App transition is allowed only if ρ is not empty, meaning that at least one AppFun has been done before. Finally, we can conclude the term building process with transition Restart only if $\kappa = 0$, meaning that all the variables of the term are bound, and if ρ is empty, meaning that there is no application waiting to be finished.

Example 5.3. Figure 5 presents how we generate the term $\lambda.\lambda.(\lambda.\underline{0})\ (1\ \lambda.0)$; we start with the underlined 0.

Any closed term t can be generated with the AB machine, and it is possible to define the sequence of flags $\mathsf{Seq}(t)$ that will be raised. We write () for the empty sequence, and $(F_1, \ldots, F_n, (F'_1, \ldots, F'_m), F_{n+1}, \ldots, F_l)$ for the sequence $(F_1, \ldots, F_n, F'_1, \ldots, F'_m, F_{n+1}, \ldots, F_l)$.

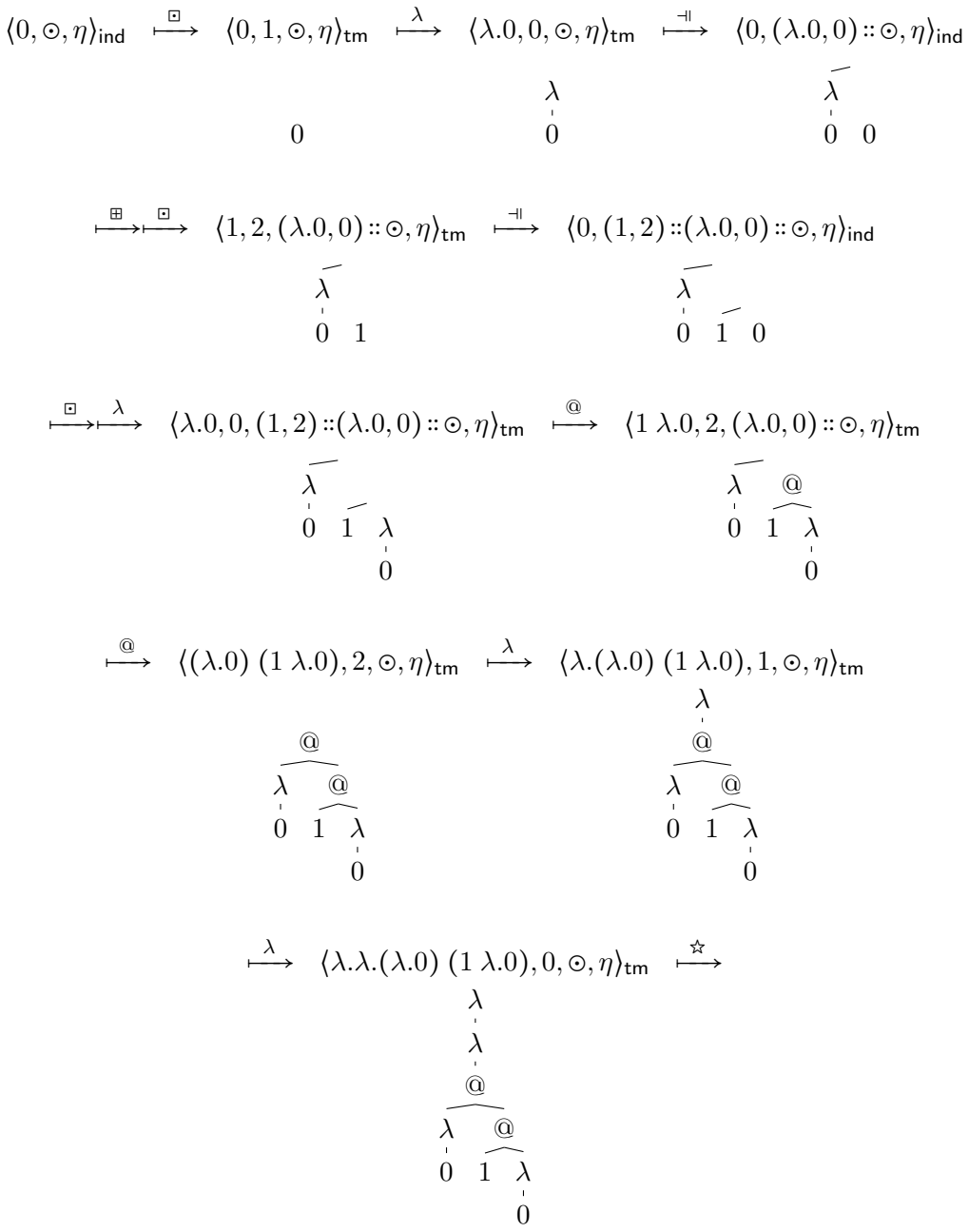

Figure 5: Example of argument generation

Definition 5.4. Given a term t, we define $\mathsf{Seq}(t)$ as

$$\mathsf{Seq}(t) \triangleq (\mathsf{SeqTm}(t), \bigstar) \qquad \mathsf{SeqTm}(n) \triangleq (\mathsf{SeqInd}(n), \boxdot)$$

$$\mathsf{SeqTm}(t\ s) \triangleq (\mathsf{SeqTm}(t), \dashv\!\mid, \mathsf{SeqTm}(s), @) \qquad \mathsf{SeqInd}(0) \triangleq ()$$

$$\mathsf{SeqTm}(\lambda.t) \triangleq (\mathsf{SeqTm}(t), \lambda) \qquad \mathsf{SeqInd}(n+1) \triangleq (\mathsf{SeqInd}(n), \boxplus)$$

We write $C \overset{\mathsf{Seq}(t)}{\longmapsto} C'$ for $C \overset{F_1}{\longmapsto} \dots \overset{F_m}{\longmapsto} C'$ where $\mathsf{Seq}(t) = (F_1, \dots, F_m)$.

Lemma 5.5. *If t' is closed, then $\langle 0, \odot, (t, e)\rangle \overset{\mathsf{Seq}(t')}{\longmapsto} \langle t, (t', \epsilon) :: e, []\rangle_{\mathsf{ev}}$.*

Proof. Let $\eta = (t, e)$. First, for all n, we have $\langle 0, \rho, \eta\rangle_{\mathsf{ind}} \overset{\boxplus\ n\ \boxdot}{\longmapsto} \langle n, n+1, \rho, \eta\rangle_{\mathsf{tm}}$. Then we show by induction on t' that $\langle 0, \rho, \eta\rangle_{\mathsf{ind}} \overset{\mathsf{SeqTm}(t')}{\longmapsto} \langle t', \kappa, \rho, \eta\rangle_{\mathsf{tm}}$ where $\kappa = \max(\mathsf{fv}(t')) + 1$ if t' is not closed, and $\kappa = 0$ otherwise. The case $t = n$ is concluded with the previous observation. If $t' = t'_1\ t'_2$, then

$$\begin{aligned}
\langle 0, \rho, \eta\rangle_{\mathsf{ind}} &\overset{\mathsf{SeqTm}(t'_1)}{\longmapsto} \langle t'_1, \kappa_1, \rho, \eta\rangle_{\mathsf{tm}} \text{ (by induction)} \\
&\overset{\dashv\shortmid}{\longmapsto} \langle 0, (t'_1, \kappa_1) :: \rho, \eta\rangle_{\mathsf{ind}} \\
&\overset{\mathsf{SeqTm}(t'_2)}{\longmapsto} \langle t'_2, \kappa_2, (t'_1, \kappa_1) :: \rho, \eta\rangle_{\mathsf{tm}} \text{ (by induction)} \\
&\longmapsto @\langle t'_1\ t'_2, \max(\kappa_1, \kappa_2), \rho, \eta\rangle_{\mathsf{tm}}
\end{aligned}$$

By case analysis on (κ_1, κ_2), one can check that $\max(\kappa_1, \kappa_2)$ is the desired value. If $t' = \lambda.t''$, then by induction, we have $\langle 0, \rho, \eta\rangle_{\mathsf{ind}} \overset{\mathsf{SeqTm}(t'')}{\longmapsto} \langle t'', \kappa, \rho, \eta\rangle_{\mathsf{tm}}$, and then $\langle t'', \kappa, \rho, \eta\rangle_{\mathsf{tm}} \overset{\lambda}{\longmapsto} \langle t', \kappa', \rho, \eta\rangle_{\mathsf{tm}}$ where κ' is as wished depending on κ.

This implies that for a closed term t', we have $\langle 0, \odot, \eta\rangle_{\mathsf{ind}} \overset{\mathsf{SeqTm}(t')}{\longmapsto} \langle t', 0, \odot, \eta\rangle_{\mathsf{tm}}$, and the last transition $\overset{\star}{\longmapsto}$ gives what we want. □

This lemma allows us to prove the correspondence between the AB machine and applicative bisimilarity.

Theorem 5.6. *$(t, e) \approx_{\mathsf{app}} (s, d)$ iff $\langle t, e, []\rangle_{\mathsf{ev}} \approx_{\mathsf{m}} \langle s, d, []\rangle_{\mathsf{ev}}$.*

Proof. To prove that machine bisimilarity implies applicative bisimilarity, we show that

$$\mathcal{R} \triangleq \{((t, e), (s, d)) \mid \langle t, e, []\rangle_{\mathsf{ev}} \approx_{\mathsf{m}} \langle s, d, []\rangle_{\mathsf{ev}}\}$$

is an applicative bisimulation. Suppose $\langle t, e, []\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*} \langle \lambda.t', e', []\rangle_{\mathsf{ev}}$. Because we have also $\langle \lambda.t', e', []\rangle_{\mathsf{ev}} \overset{\circledcirc}{\longmapsto} \langle 0, \odot, (t', e')\rangle_{\mathsf{ind}}$, there exist s' and d' such that

$$\langle s, d, []\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*} \langle \lambda.s', d', []\rangle_{\mathsf{ev}} \overset{\circledcirc}{\longmapsto} \langle 0, \odot, (s', d')\rangle_{\mathsf{ind}}$$

and $\langle 0, \odot, (t', e')\rangle_{\mathsf{ind}} \approx_{\mathsf{m}} \langle 0, \odot, (s', d')\rangle_{\mathsf{ind}}$. Let t'' be a closed term. By Lemma 5.5, we have $\langle 0, \odot, (t', e')\rangle \overset{\mathsf{Seq}(t'')}{\longmapsto} \langle t', (t'', \epsilon) :: e', []\rangle_{\mathsf{ev}}$, which can only be matched by $\langle 0, \odot, (s', d')\rangle \overset{\mathsf{Seq}(t'')}{\longmapsto} \langle s', (t'', \epsilon) :: d', []\rangle_{\mathsf{ev}}$. From $\langle 0, \odot, (t', e')\rangle_{\mathsf{ind}} \approx_{\mathsf{m}} \langle 0, \odot, (s', d')\rangle_{\mathsf{ind}}$, we get $\langle t', (t'', \epsilon) :: e', []\rangle_{\mathsf{ev}} \approx_{\mathsf{m}} \langle s', (t'', \epsilon) :: d', []\rangle_{\mathsf{ev}}$. It implies that $(t', (t'', \epsilon) :: e')\ \mathcal{R}\ (s', (t'', \epsilon) :: d')$ holds, as wished.

For the reverse implication, we show that

$$\begin{aligned}
\mathcal{R} \triangleq\ & \{(\langle t, e, []\rangle_{\mathsf{ev}}, \langle s, d, []\rangle_{\mathsf{ev}}) \mid (t, e) \approx_{\mathsf{app}} (s, d)\} \\
& \cup \{(\langle n, \rho, (t, e)\rangle_{\mathsf{ind}}), \langle n, \rho, (s, d)\rangle_{\mathsf{ind}}) \mid (\lambda.t, e) \approx_{\mathsf{app}} (\lambda.s, d)\} \\
& \cup \{(\langle t', \kappa, \rho, (t, e)\rangle_{\mathsf{tm}}), \langle t', \kappa, \rho, (s, d)\rangle_{\mathsf{tm}}) \mid (\lambda.t, e) \approx_{\mathsf{app}} (\lambda.s, d)\}
\end{aligned}$$

is a machine bisimulation. Suppose $\langle t, e, []\rangle_{\mathsf{ev}}\ \mathcal{R}\ \langle s, d, []\rangle_{\mathsf{ev}}$. The only possible flagged transition is $\langle t, e, []\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*} \overset{\circledcirc}{\longmapsto} \langle 0, \odot, (t', e')\rangle_{\mathsf{ind}}$, which implies in particular $\langle t, e, []\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*} \langle \lambda.t', e', []\rangle_{\mathsf{ev}}$. Because $(t, e) \approx_{\mathsf{app}} (s, d)$, there exists (s', d') such that $\langle s, d, []\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*}$

$\langle \lambda.s', d', [] \rangle_{\mathsf{ev}}$ and $(\lambda.t', e') \approx_{\mathsf{app}} (\lambda.s', d')$. We also have $\langle \lambda.s', d', [] \rangle_{\mathsf{ev}} \overset{@}{\longmapsto} \langle 0, \odot, (s', d') \rangle_{\mathsf{ind}}$, hence $\langle s, d, [] \rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*} \overset{@}{\longmapsto} \langle 0, \odot, (s', d') \rangle_{\mathsf{ind}}$ and $\langle 0, \odot, (t', e') \rangle_{\mathsf{ind}} \mathcal{R} \langle 0, \odot, (s', d') \rangle_{\mathsf{ind}}$ hold, as wished.

To cover the last two sets of the bisimulation, let $\langle t', \kappa, \rho, (t, e) \rangle_{\mathsf{arg}} \mathcal{R} \langle t', \kappa, \rho, (s, d) \rangle_{\mathsf{arg}}$ with $\mathsf{arg} \in \{\mathsf{ind}, \mathsf{tm}\}$. If $\langle t', \kappa, \rho, (t, e) \rangle_{\mathsf{arg}} \overset{F}{\longmapsto} C$ with $F \neq \bigstar$, then $\langle t', \kappa, \rho, (s, d) \rangle_{\mathsf{arg}} \overset{F}{\longmapsto} C'$ where C' is the same as C except for the closures. These are not changed by the transition and are still applicative bisimilar; we have therefore $C \mathcal{R} C'$. Otherwise, we have $\langle t', \kappa, \rho, (t, e) \rangle \overset{\bigstar}{\longmapsto} \langle t, (t', \epsilon) :: e, [] \rangle_{\mathsf{ev}}$, and also $\langle t', \kappa, \rho, (s, d) \rangle \overset{\bigstar}{\longmapsto} \langle s, (t', \epsilon) :: d, [] \rangle_{\mathsf{ev}}$. The resulting configurations are in $\mathcal{R}$ because $(\lambda.t, e) \approx_{\mathsf{app}} (\lambda.s, d)$ implies $(t, (t', \epsilon) :: e) \approx_{\mathsf{app}} (s, (t', \epsilon) :: d)$ for all closed t'. □

5.3. **Translation into HOcore.** We detail each component of the translation of the AB machine into HOcore, starting with the evaluation mode, i.e., the KAM.

$$[\![\langle t, e, \pi \rangle_{\mathsf{ev}}]\!] \triangleq [\![t]\!] \parallel \overline{env}\langle [\![e]\!] \rangle \parallel \overline{c}\langle [\![\pi]\!] \rangle \parallel P_{\mathsf{rec}} \qquad [\![(t, e)]\!] \triangleq \overline{\eta_1}\langle [\![t]\!] \rangle \parallel \overline{\eta_2}\langle [\![e]\!] \rangle$$

$$[\![\eta :: \pi]\!] \triangleq \overline{hd_{\mathsf{c}}}\langle [\![\eta]\!] \rangle \parallel \overline{c}\langle [\![\pi]\!] \rangle \qquad [\![\eta :: e]\!] \triangleq \overline{hd_{\mathsf{e}}}\langle [\![\eta]\!] \rangle \parallel \overline{env}\langle [\![e]\!] \rangle$$

$$[\![[]]\!] \triangleq \ldots \qquad [\![\epsilon]\!] \triangleq \mathbf{0}$$

We follow the same principles as in Section 3: a non-empty stack π or environment e is represented by a pair of messages, respectively on hd_{c} and c, and hd_{e} and env (we use the longer env for legibility reasons). A closure is represented by two messages, one containing the term on η_1 and the other the environment on η_2. The process representing the empty environment ϵ should never be executed, because all the closures we manipulate are closed; as a result, we can choose any process to represent it, e.g., $\mathbf{0}$. The empty stack $[\![[]]\!]$ and the process P_{rec} are used to generate an argument and are defined and explained later.

$$[\![t\ s]\!] \triangleq \mathsf{App}_{\mathsf{ev}}([\![t]\!], [\![s]\!])$$

$$\mathsf{App}_{\mathsf{ev}}(P_t, P_s) \triangleq c(x).env(y).(P_t \parallel \overline{c}\Big\langle \overline{hd_{\mathsf{c}}}\langle \overline{\eta_1}\langle P_s \rangle \parallel \overline{\eta_2}\langle y \rangle \rangle \parallel x \Big\rangle \parallel \overline{env}\langle y \rangle)$$

$$[\![\lambda.t]\!] \triangleq \mathsf{Lam}_{\mathsf{ev}}([\![t]\!])$$

$$\mathsf{Lam}_{\mathsf{ev}}(P_t) \triangleq c(x).(x \parallel hd_{\mathsf{c}}(y).env(z).(P_t \parallel \overline{env}\Big\langle \overline{hd_{\mathsf{e}}}\langle y \rangle \parallel \overline{env}\langle z \rangle \Big\rangle))$$

$$[\![n+1]\!] \triangleq \mathsf{Ind}_{\mathsf{ev}}([\![n]\!])$$

$$\mathsf{Ind}_{\mathsf{ev}}(P_n) \triangleq env(x).(x \parallel hd_{\mathsf{e}}(\_).P_n)$$

$$[\![0]\!] \triangleq env(x).(x \parallel hd_{\mathsf{e}}(y).env(\_).(y \parallel \eta_1(y_1).\eta_2(y_2).(y_1 \parallel \overline{env}\langle y_2 \rangle)))$$

The encoding of $t\ s$ simulates the rule Push: we receive the current stack and environment e to create the new stack with (s, e) on top. Because we receive the current environment to put it on the stack, we have to recreate it on env, unchanged. In the encoding of $\lambda.t$, we capture the stack and environment, and if the stack is non-empty, we fetch its head η to create a new environment with η on top. Finally, a de Bruijn index $n > 0$ goes through the current environment, until we reach the correct closure (case $n = 0$). In that case, we receive the head η and tail of the environment, we discard the tail as it is no longer useful, and we restore the term and environment stored in η.

If $\lambda.t$ is run in the environment e and the empty stack $[]$, then we obtain $[\![[]]\!] \parallel \mathsf{Stuck}((t,e))$, where $\mathsf{Stuck}((t,e)) \triangleq hd_{\mathsf{c}}(z).([\![t]\!] \parallel \overline{env}\langle \overline{hd_{\mathsf{e}}}\langle z\rangle \parallel \overline{env}\langle [\![e]\!]\rangle\rangle)$. The process $[\![[]]\!]$ has to start the argument generating process, and the result has then to be sent on hd_{c} for the evaluation to restart. The process $\mathsf{Stuck}((t,e))$ remains stuck during the whole generation process. We now explain how $\langle n,\rho,\eta\rangle_{\mathsf{ind}}$ and $\langle t,\kappa,\rho,\eta\rangle_{\mathsf{tm}}$ are encoded, starting with κ and ρ.

$$[\![0]\!]_{\mathsf{c}} \triangleq zero(x).suc(\_).x \qquad [\![\odot]\!] \triangleq mt(x).cs(\_).x$$
$$[\![\kappa+1]\!]_{\mathsf{c}} \triangleq \mathsf{Suk}([\![\kappa]\!]_{\mathsf{c}}) \qquad [\![(t,\kappa)::\rho]\!] \triangleq \mathsf{ConsR}([\![(t,\kappa)]\!],[\![\rho]\!])$$
$$\mathsf{Suk}(P_\kappa) \triangleq \overline{suk}\langle P_\kappa\rangle \parallel zero(\_).suc(x).x \qquad \mathsf{ConsR}(P_{hd},P_\rho) \triangleq \overline{hd_{\mathsf{r}}}\langle P_{hd}\rangle \parallel \overline{r}\langle P_\rho\rangle \parallel mt(\_).cs(x).x$$
$$[\![(t,\kappa)]\!] \triangleq \overline{w_1}\langle [\![t]\!]\rangle \parallel \overline{w_2}\langle [\![\kappa]\!]_{\mathsf{c}}\rangle$$

The machine distinguishes cases based on whether κ is 0 or not, to know if we should apply the transition Lambda or Lambda0. In the encoding of these rules (see the definition of Lambda below), we send on name *zero* the expected behavior if $\kappa = 0$, and on *suc* what to do otherwise. The translation of the counter receives both messages, executes the corresponding one (e.g., the one on *zero* for the encoding of 0), and discards the other. Apart from that, κ is translated as nested sending on name *suk*. Similarly, the translation of ρ combines the regular encodings of pairs and stacks, but also indicates whether ρ is empty or not, to know if we can apply the transitions App and Restart.

$$[\![[]]\!] \triangleq \circledcirc(\_).(\overline{ind}\langle [\![0]\!]\rangle \parallel \overline{k}\langle [\![1]\!]_{\mathsf{c}}\rangle \parallel \overline{r}\langle [\![\odot]\!]\rangle \parallel \overline{initInd}\langle \mathbf{0}\rangle)$$

$$[\![\langle n,\rho,\eta\rangle_{\mathsf{ind}}]\!] \triangleq \overline{ind}\langle [\![n]\!]\rangle \parallel \overline{k}\langle [\![n+1]\!]_{\mathsf{c}}\rangle \parallel \overline{r}\langle [\![\rho]\!]\rangle \parallel P_{\mathsf{rec}} \parallel \overline{initInd}\langle \mathbf{0}\rangle \parallel \mathsf{Stuck}(\eta)$$
$$\mathsf{RecInd} \triangleq initInd(\_).recind(x).(x \parallel \overline{recind}\langle x\rangle \parallel \mathsf{Succ}+\mathsf{Var})$$
$$\mathsf{Succ} \triangleq \boxplus(\_).ind(x).k(y).(\overline{ind}\langle \mathsf{Ind}_{\mathsf{ev}}(x)\rangle \parallel \overline{k}\langle \mathsf{Suk}(y)\rangle \parallel \overline{initInd}\langle \mathbf{0}\rangle)$$
$$\mathsf{Var} \triangleq \boxdot(\_).ind(x).(\overline{tm}\langle x\rangle \parallel \overline{initTm}\langle \mathbf{0}\rangle)$$

$$[\![\langle t,\kappa,\rho,\eta\rangle_{\mathsf{tm}}]\!] \triangleq \overline{tm}\langle [\![t]\!]\rangle \parallel \overline{k}\langle [\![\kappa]\!]_{\mathsf{c}}\rangle \parallel \overline{r}\langle [\![\rho]\!]\rangle \parallel P_{\mathsf{rec}} \parallel \overline{initTm}\langle \mathbf{0}\rangle \parallel \mathsf{Stuck}(\eta)$$

After flagging $\circledcirc$, the process $[\![[]]\!]$ starts the argument generation process in index mode: the index being built is sent on *ind* (here, initialized with $[\![0]\!]$) and the stack on r. We also build on k the counter κ corresponding to the index: at any point, we have $\kappa = n+1$. We keep two messages on *ind* and k encoding almost the same information, but with different encodings, as the index n and the counter κ are used differently.

The message on *initInd* triggers the recursive process RecInd, which non-deterministically chooses between Succ and Var. Executing Succ flags $\boxplus$, increases the values of the index (thanks to $\mathsf{Ind}_{\mathsf{ev}}$) and the counter (with Suk), and relaunches the RecInd process with a message on *initInd*. Executing Var flags $\boxdot$, moves the index from *ind* to *tm*, and initiates

the term mode by sending a message on $initTm$, which triggers the recursive process RecTm.

$$\mathsf{RecTm} \triangleq initTm(\_).rectm(x).$$
$$\left(\begin{array}{l} x \parallel \overline{rectm}\langle x\rangle \parallel r(y).y \parallel \overline{cs}\left\langle \begin{array}{l} k(z).hd_{\mathsf{r}}(y_1).r(y_2). \\ (\mathsf{Lambda}(z) \parallel \overline{r}\langle y\rangle) + \mathsf{AppFun}(z,y) + \mathsf{App}(z,y_1,y_2) \end{array}\right\rangle \\ \qquad \parallel \overline{mt}\left\langle k(z).\left(\begin{array}{l} z \parallel \overline{zero}\left\langle \begin{array}{r} (\mathsf{Lambda}(z) \parallel \overline{r}\langle[\![\odot]\!]\rangle) \\ + \mathsf{AppFun}(z,[\![\odot]\!]) + \mathsf{Done} \end{array}\right\rangle \\ \quad \parallel \overline{suc}\left\langle suk(\_). \begin{array}{r} (\mathsf{Lambda}(z) \parallel \overline{r}\langle[\![\odot]\!]\rangle) \\ + \mathsf{AppFun}(z,[\![\odot]\!]) \end{array}\right\rangle \end{array}\right)\right\rangle \end{array}\right)$$

The goal of RecTm is to non-deterministically choose between the four transitions available in term mode, namely $\overset{\lambda}{\longmapsto}$, $\overset{\dashv\!\shortparallel}{\longmapsto}$, $\overset{@}{\longmapsto}$, and $\overset{\bigstar}{\longmapsto}$. However, some of these transitions have requirements: $\overset{@}{\longmapsto}$ needs $\rho \neq \odot$ and $\overset{\bigstar}{\longmapsto}$ needs $\rho = \odot$ and $\kappa = 0$. The process RecTm is therefore doing a case analysis to check these conditions. First, it captures $[\![\rho]\!]$ on r: if $\rho \neq \odot$, it executes the message on cs, which makes a choice between λ, $\dashv\!\shortparallel$, and $@$, which are represented by respectively Lambda, AppFun, and App. If $\rho = \odot$, then we do a case analysis on κ. If $\kappa = 0$, then we can do either λ, $\dashv\!\shortparallel$, or $\bigstar$ (represented by Done), otherwise, only λ or $\dashv\!\shortparallel$ are possible.

$$\mathsf{Lambda}(P_\kappa) \triangleq \lambda(\_).tm(x).\left(\begin{array}{l} \overline{tm}\langle \mathsf{Lam}_{\mathsf{ev}}(x)\rangle \parallel P_\kappa \parallel \overline{zero}\left\langle \overline{k}\langle P_\kappa\rangle \parallel \overline{initTm}\langle \mathbf{0}\rangle\right\rangle \\ \qquad\qquad \parallel \overline{suc}\left\langle suk(y).(\overline{k}\langle y\rangle \parallel \overline{initTm}\langle \mathbf{0}\rangle)\right\rangle \end{array}\right)$$

$$\mathsf{AppFun}(P_\kappa, P_\rho) \triangleq \dashv\!\shortparallel(\_).tm(x).\left(\begin{array}{l} \overline{r}\left\langle \mathsf{ConsR}(\overline{hd_{\mathsf{r}}}\left\langle \overline{w_1}\langle x\rangle \parallel \overline{w_2}\langle P_\kappa\rangle\right\rangle, P_\rho)\right\rangle \parallel \\ \overline{ind}\langle[\![0]\!]\rangle \parallel \overline{k}\langle[\![1]\!]_{\mathsf{c}}\rangle \parallel \overline{initInd}\langle \mathbf{0}\rangle \end{array}\right)$$

$$\mathsf{App}(P_\kappa, P_{hd}, P_\rho) \triangleq @(\_).tm(x_2).$$
$$\left(\begin{array}{l} P_{hd} \parallel w_2(y).w_1(x_1).\left(\begin{array}{l} \overline{max1}\langle y\rangle \parallel \overline{max2}\langle P_\kappa\rangle \parallel \overline{init1}\langle y\rangle \parallel \overline{init2}\langle P_\kappa\rangle \parallel \\ resu(z).\left(\begin{array}{l} \overline{tm}\langle \mathsf{App}_{\mathsf{ev}}(x_1,x_2)\rangle \parallel \overline{r}\langle P_\rho\rangle \parallel \\ \overline{k}\langle z\rangle \parallel \overline{initTm}\langle \mathbf{0}\rangle \end{array}\right) \end{array}\right) \end{array}\right)$$

The process Lambda adds a λ-abstraction to the term in tm, updating κ (represented by P_κ) accordingly: if $\kappa = 0$, then it is restored unchanged on k, otherwise, it is decreased by 1 by releasing the message in suk. The process AppFun pushes on the stack P_ρ the current term t_1 on tm and its counter κ_1 (represented by P_κ), which is the term in function position of an application. It then relaunches the index mode to build the argument t_2 with its counter κ_2. The process App can then build the application itself, by computing the maximum between κ_1 and κ_2 with the processes RecMax and Max.

$$\mathsf{RecMax} \triangleq init1(x_1).init2(x_2).recmax(y).(y \parallel \overline{recmax}\langle y\rangle \parallel \mathsf{Max}(x_1,x_2))$$
$$\mathsf{Max}(P_1,P_2) \triangleq P_1 \parallel \overline{zero}\left\langle max2(x).\overline{resu}\langle x\rangle\right\rangle$$
$$\parallel \overline{suc}\left\langle suk(x_1).\left(\begin{array}{l} P_2 \parallel \overline{zero}\left\langle max1(x).\overline{resu}\langle x\rangle\right\rangle \\ \quad \parallel \overline{suc}\left\langle suk(x_2).(\overline{init1}\langle x_1\rangle \parallel \overline{init2}\langle x_2\rangle)\right\rangle \end{array}\right)\right\rangle$$

We compute the maximum between κ_1 and κ_2 by removing the layers of successors common to κ_1 and κ_2, until we reach 0 for one of them. If we reach 0 for κ_1 first, then κ_2 is the max,

otherwise it is κ_1. We store the initial values of κ_1 and κ_2 in respectively $max1$ and $max2$, and the decomposition occurs in Max, where $init1$ is initialized with κ_1 and $init2$ with κ_2. If $P_1 = [\![\kappa_1]\!]_{\mathsf{c}} = [\![0]\!]_{\mathsf{c}}$, then we send κ_2 (stored in $max2$) on $resu$. Otherwise, $P_1 = \overline{suk}\langle P_1' \rangle$, and we do a case analysis on the process $P_2 = [\![\kappa_2]\!]_{\mathsf{c}}$. If $P_2 = [\![0]\!]_{\mathsf{c}}$, then we send κ_1 on $resu$, otherwise $P_2 = \overline{suk}\langle P_2' \rangle$, and we restart RecMax by sending P_1' and P_2' on $init1$ and $init2$, respectively. Once the max is known on $resu$, then App builds the application $t_1\ t_2$ and relaunches RecTm.

$$\mathsf{Done} \triangleq \star(\_).tm(x).(\overline{hd_{\mathsf{c}}}\langle \overline{\eta_1}\langle x\rangle \parallel \overline{\eta_2}\langle [\![\epsilon]\!]\rangle\rangle \parallel \overline{c}\langle [\![[\,]]\!]\rangle)$$

$$P_{\mathsf{rec}} \triangleq \mathsf{RecInd} \parallel \overline{recind}\langle \mathsf{RecInd}\rangle \parallel \mathsf{RecTm} \parallel \overline{rectm}\langle \mathsf{RecTm}\rangle \parallel \mathsf{RecMax} \parallel \overline{recmax}\langle \mathsf{RecMax}\rangle$$

Finally, the process Done ends the argument generation phase, and restarts the computation by restoring the empty continuation and by passing the term in tm to $\mathsf{Stuck}(\eta)$. The process P_{rec} contains all the processes necessary to encode the different recursive mechanisms.

Full abstraction. A case analysis on the configurations of the machine shows that the translation of the AB machine satisfies the conditions of Section 4.4, and we can prove a full abstraction result between AB machines and HOcore similar to Theorem 4.14. We deduce a full abstraction result between the λ-calculus with applicative bisimilarity and HOcore.

Corollary 5.7. *For all closed closures* (t, e) *and* (s, d), $(t, e) \approx_{\mathsf{app}} (s, d)$ *iff* $[\![\langle t, e, [\,]\rangle_{\mathsf{ev}}]\!] \approx_{\mathsf{HO}} [\![\langle s, d, [\,]\rangle_{\mathsf{ev}}]\!]$.

5.4. **Internalizing Contextual Equivalence.** Corollary 5.7 is enough to deduce full abstraction w.r.t. contextual equivalence, since $t \approx_{\mathsf{ctx}} s \iff (t, \epsilon) \approx_{\mathsf{app}} (s, \epsilon)$. However, it is possible to prove a similar result directly, by internalizing contextual equivalence in an abstract machine. We do so to show how the resulting contextual equivalence machine differs from the AB machine.

Figure 6 gives the transitions of the contextual equivalence machine, except for the $\overset{\tau}{\longmapsto}$ transitions, which are the same as in Section 5.1. In contrast with the AB machine, the contextual equivalence machine produces a context first, and then reduces the resulting term; consequently, the starting point is a state $\langle t, 0, \odot\rangle_{\mathsf{tm}}$, where t is the closed term we want to plug in the context. When the context is finished, the transition $\overset{\circledast}{\longmapsto}$ switches to the evaluation mode. Also, the evaluation part of the machine is not executed several times, since $\approx_{\mathsf{ctx}}$ is not coinductive. We flag ☆ when the evaluation terminates, to distinguish a terminating term from a diverging one.

Creating a context $\mathcal{C}$ is almost the same as generating an argument in the AB machine, except that we want to plug a closed term t inside. We build $\mathcal{C}[t]$ by starting the generation process from t; t can be anywhere in $\mathcal{C}[t]$, not necessarily at the leftmost position, so we cannot do the generation process going left to right in an application, as with the AB machine (Section 5.2). Instead, after producing a term t and with a term s on the stack, we can do either the transition AppLeft to build $t\ s$, or AppRight to build $s\ t$.

$$\langle n, \kappa, \rho\rangle_{\mathsf{ind}} \overset{\boxplus}{\longmapsto} \langle n+1, \kappa+1, \rho\rangle_{\mathsf{ind}} \qquad \text{(Suc)}$$

$$\langle n, \kappa, \rho\rangle_{\mathsf{ind}} \overset{\boxdot}{\longmapsto} \langle n, \kappa, \rho\rangle_{\mathsf{tm}} \qquad \text{(Var)}$$

$$\langle t, \kappa+1, \rho\rangle_{\mathsf{tm}} \overset{\lambda}{\longmapsto} \langle \lambda.t, \kappa, \rho\rangle_{\mathsf{tm}} \qquad \text{(Lambda)}$$

$$\langle t, 0, \rho\rangle_{\mathsf{tm}} \overset{\lambda}{\longmapsto} \langle \lambda.t, 0, \rho\rangle_{\mathsf{tm}} \qquad \text{(Lambda0)}$$

$$\langle t, \kappa, \rho\rangle_{\mathsf{tm}} \overset{\dashv\!\shortmid}{\longmapsto} \langle 0, 1, (t,\kappa) :: \rho\rangle_{\mathsf{ind}} \qquad \text{(AppPush)}$$

$$\langle t, \kappa_1, (s,\kappa_2) :: \rho\rangle_{\mathsf{tm}} \overset{\overleftarrow{@}}{\longmapsto} \langle t\ s, \mathsf{max}(\kappa_1,\kappa_2), \rho\rangle_{\mathsf{tm}} \qquad (\overleftarrow{\text{App}})$$

$$\langle t, \kappa_1, (s,\kappa_2) :: \rho\rangle_{\mathsf{tm}} \overset{\overrightarrow{@}}{\longmapsto} \langle s\ t, \mathsf{max}(\kappa_1,\kappa_2), \rho\rangle_{\mathsf{tm}} \qquad (\overrightarrow{\text{App}})$$

$$\langle t, 0, \odot\rangle_{\mathsf{tm}} \overset{\circledast}{\longmapsto} \langle t, \epsilon, [\,]\rangle_{\mathsf{ev}} \qquad \text{(Start)}$$

$$\langle \lambda.t, e, [\,]\rangle_{\mathsf{ev}} \overset{\star}{\longmapsto} \qquad \text{(Done)}$$

Figure 6: Contextual equivalence machine

Example 5.8. We show how to generate the context $\lambda.(0\ 0)\ (\square\ 0)$ around t.

$$\begin{aligned}
\langle t, 0, \odot\rangle_{\mathsf{tm}} &\overset{\dashv\!\shortmid}{\longmapsto} \overset{\boxdot}{\longmapsto} \langle 0, 1, (t,0) :: \odot\rangle_{\mathsf{tm}} \\
&\overset{\overrightarrow{@}}{\longmapsto} \langle t\ 0, 1, \odot\rangle_{\mathsf{tm}} \\
&\overset{\dashv\!\shortmid}{\longmapsto} \overset{\boxdot}{\longmapsto} \langle 0, 1, (t\ 0, 1) :: \odot\rangle_{\mathsf{tm}} \\
&\overset{\dashv\!\shortmid}{\longmapsto} \overset{\boxdot}{\longmapsto} \langle 0, 1, (0,1) :: (t\ 0, 1) :: \odot\rangle_{\mathsf{tm}} \\
&\overset{\overrightarrow{@}}{\longmapsto} \langle 0\ 0, 1, (t\ 0, 1) :: \odot\rangle_{\mathsf{tm}} \\
&\overset{\overleftarrow{@}}{\longmapsto} \langle (0\ 0)\ (t\ 0), 1, \odot\rangle_{\mathsf{tm}} \\
&\overset{\lambda}{\longmapsto} \langle \lambda.(0\ 0)\ (t\ 0), 0, \odot\rangle_{\mathsf{tm}}
\end{aligned}$$

The translation of the contextual equivalence machine into HOcore and the full abstraction proofs are similar to the AB machine ones.

Theorem 5.9. *If t and s are closed terms, then $t \approx_{\mathsf{ctx}} s$ iff $[\![\langle t, 0, \odot\rangle_{\mathsf{tm}}]\!] \approx_{\mathsf{HO}} [\![\langle s, 0, \odot\rangle_{\mathsf{tm}}]\!]$.*

6. Call-by-Value

We adapt our techniques to the call-by-value setting, by internalizing equivalences into the CK machine.

6.1. **The CK machine.** As in call-by-name, we internalize normal-form bisimilarity into a substitution-based abstract machine, namely the CK machine [FFF09].

$$C ::= \langle t, \pi\rangle \mid \langle \pi, v\rangle_{\mathsf{ct}} \qquad \text{(configurations)}$$
$$v ::= x \mid \lambda x.t \qquad \text{(values)}$$
$$t, s ::= v \mid t\ s \qquad \text{(terms)}$$
$$\pi ::= \bullet\ t :: \pi \mid v\ \bullet :: \pi \mid [] \qquad \text{(stacks)}$$

$$\langle t\ s, \pi\rangle \mapsto \langle t, \bullet\ s :: \pi\rangle \qquad \text{(Fun)}$$
$$\langle v, \pi\rangle \mapsto \langle \pi, v\rangle_{\mathsf{ct}} \qquad \text{(Switch)}$$
$$\langle \bullet\ t :: \pi, v\rangle_{\mathsf{ct}} \mapsto \langle t, v\ \bullet :: \pi\rangle \qquad \text{(Arg)}$$
$$\langle \lambda x.t\ \bullet :: \pi, v\rangle_{\mathsf{ct}} \mapsto \langle [v\,/\,x]t, \pi\rangle \qquad \text{(Beta)}$$

The machine distinguishes configurations of the form $\langle t, \pi\rangle$, which evaluates t in the stack π, from configurations $\langle \pi, v\rangle_{\mathsf{ct}}$, which decide how computation should proceed depending on the stack π. In left-to-right call-by-value evaluation, we evaluate the function before reducing its argument. What we remember on the stack reflects this order: $\bullet\ t$ means that we evaluate the function and remembers its argument t, while $v\ \bullet$ means that we evaluate the argument of an already computed function v.

The Fun machine step focuses on the term in function position. When it is evaluated, we look at the stack by switching to the continuation mode (rule Switch). Either the top of the stack is an argument which needs to be evaluated (rule Arg), or it is a λ-abstraction ready to be applied to its argument (rule Beta). When we reduce closed terms, we cannot get a configuration of the form $\langle x\ \bullet :: \pi, v\rangle_{\mathsf{ct}}$.

The encoding of the CK machine in HOcore is given in Figure 7. Unlike in the KAM, the stack has some control on how the reduction proceeds. It is reflected in the encoding of $\langle \pi, v\rangle_{\mathsf{ct}}$, where the encoded value $[\![v]\!]_{\mathsf{v}}$ is sent on a channel name val, waiting to be consumed by the stack. The top of the stack decides what becomes of $[\![v]\!]_{\mathsf{v}}$: in the case of $\bullet\ t$, the value $[\![v]\!]_{\mathsf{v}}$ is forwarded to the top of the stack and $[\![t]\!]$ can reduce. For $\lambda x.t\ \bullet$, the encoded λ-abstraction $val(x).[\![t]\!]$ is receiving $[\![v]\!]_{\mathsf{v}}$, and the computation continues as $[[\![v]\!]_{\mathsf{v}}\,/\,x]\,[\![t]\!]$: as for the KAM, we encode β-reduction by a HOcore communication.

The encoding of values considered as terms $[\![v]\!]$ is uniform, and recreates the encoding of $\langle \pi, v\rangle_{\mathsf{ct}}$ by capturing the current stack and running it in parallel with $\overline{val}\langle [\![v]\!]_{\mathsf{v}}\rangle$. The encoding of the application $[\![t\ s]\!]$ is as expected: it runs $[\![t]\!]$ and pushes $[\![s]\!]$ on the current stack. Like for the KAM, the encoding of the CK machine needs only two names, c and val.

We have a one-to-one correspondence between the CK-machine and its translation, a result stronger than for the KAM (cf. Theorem 3.1): the administrative Switch step, which does not exist in the KAM, corresponds to a communication in the HOcore translation.

Theorem 6.1. *If $C \mapsto C'$, then $[\![C]\!] \xrightarrow{\tau} [\![C']\!]$. If $[\![C]\!] \xrightarrow{\tau} P$, then there exists C' such that $P = [\![C']\!]$.*

6.2. **Bisimilarities.** To internalize normal-form bisimilarity, we distinguish free variables f from bound variables x, as in Section 4. We assume λ-terms to be well-formed, so that every bound variable has its binder.

$$t, s ::= f \mid x \mid \lambda x.t \mid t\ s \qquad v ::= f \mid \lambda x.t$$

$$
\begin{aligned}
[\![\langle t, \pi\rangle]\!] &\triangleq [\![t]\!] \parallel \overline{c}\langle [\![\pi]\!]\rangle \\
[\![\langle \pi, v\rangle_{\mathsf{ct}}]\!] &\triangleq [\![\pi]\!] \parallel \overline{val}\langle [\![v]\!]_{\mathsf{v}}\rangle \\
[\![[\,]]\!] &\triangleq \overline{b}\langle \mathbf{0}\rangle \\
[\![\bullet\, t :: \pi]\!] &\triangleq \mathsf{Arg}([\![t]\!], [\![\pi]\!]) \\
[\![v\ \bullet :: \pi]\!] &\triangleq \mathsf{Fun}([\![v]\!]_{\mathsf{v}}, [\![\pi]\!]) \\
\mathsf{Arg}(P_t, P_\pi) &\triangleq val(x).(P_t \parallel \overline{c}\langle \mathsf{Fun}(x, P_\pi)\rangle) \\
\mathsf{Fun}(P_v, P_\pi) &\triangleq P_v \parallel \overline{c}\langle P_\pi\rangle
\end{aligned}
$$

$$
\begin{aligned}
[\![x]\!]_{\mathsf{v}} &\triangleq x \\
[\![\lambda x.t]\!]_{\mathsf{v}} &\triangleq val(x).\,[\![t]\!] \\
[\![v]\!] &\triangleq c(p).(p \parallel \overline{val}\langle [\![v]\!]_{\mathsf{v}}\rangle) \\
[\![t\ s]\!] &\triangleq c(p).([\![t]\!] \parallel \overline{c}\langle \mathsf{Arg}([\![s]\!], p)\rangle)
\end{aligned}
$$

Figure 7: Encoding of the CK machine in HOcore

The evaluation of an open term may result either in a value in an empty stack, or in a *stuck term* of the form $\langle f\ \bullet :: \pi, v\rangle_{\mathsf{ct}}$: because the function is a free variable, the β-reduction cannot be triggered. Normal-form bisimilarity should be defined on these two kinds of normal forms.

For values, a first possibility is to proceed as in call-by-name, i.e., to equate identical free variables, and to relate the bodies of λ-abstractions. This variant (discussed in Remark 6.4) is too discriminating in the plain λ-calculus: e.g., it distinguishes a free variable f from its η-expansion $\lambda x.f\ x$ although these terms are contextually equivalent in call-by-value. Lassen [Las05] proposes instead to compare values by applying them to a fresh variable. For instance, applying f and $\lambda x.f\ x$ to a fresh variable f' produces the same term $f\ f'$ on both sides after evaluation.

Normal-form bisimilarity compares two stuck terms $\langle f\ \bullet :: \pi, v\rangle_{\mathsf{ct}}$ and $\langle f\ \bullet :: \pi', v'\rangle_{\mathsf{ct}}$ by relating v and v', and by testing π and π' with a fresh variable f'. To reflect this test on stacks, we define normal-form bisimilarity not on terms, but on configurations.

Definition 6.2. A symmetric relation $\mathcal{R}$ on configurations is a normal-form bisimulation if $C\ \mathcal{R}\ C'$ implies:

- if $C \mapsto^* \langle [\,], v\rangle_{\mathsf{ct}}$, then there exists v' such that $C' \mapsto^* \langle [\,], v'\rangle_{\mathsf{ct}}$ and for all fresh f, we have $\langle v\ \bullet :: [\,], f\rangle_{\mathsf{ct}}\ \mathcal{R}\ \langle v'\ \bullet :: [\,], f\rangle_{\mathsf{ct}}$;
- if $C \mapsto^* \langle f\ \bullet :: \pi, v\rangle_{\mathsf{ct}}$, then there exist π' and v' such that $C' \mapsto^* \langle f\ \bullet :: \pi', v'\rangle_{\mathsf{ct}}$, and for all fresh f', we have $\langle v\ \bullet :: [\,], f'\rangle_{\mathsf{ct}}\ \mathcal{R}\ \langle v'\ \bullet :: [\,], f'\rangle_{\mathsf{ct}}$ and $\langle \pi, f'\rangle_{\mathsf{ct}}\ \mathcal{R}\ \langle \pi', f'\rangle_{\mathsf{ct}}$.

Normal-form bisimilarity $\approx_{\mathsf{nf}}$, is the largest normal-form bisimulation.

In both clauses, we compare values by considering configurations of the form $\langle v\ \bullet :: [\,], f\rangle_{\mathsf{ct}}$, which corresponds to the application $v\ f$, as wished. Normal-form bisimilarity is extended to terms so that $t \approx_{\mathsf{nf}} s$ if $\langle t, [\,]\rangle \approx_{\mathsf{nf}} \langle s, [\,]\rangle$. This equivalence can be related to Böhm tree equivalence up to infinite η expansion through a continuation passing style

$$\langle t\ s, \pi, n\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto} \langle t, \bullet\ s :: \pi, n\rangle_{\mathsf{ev}} \qquad \text{(Fun)}$$

$$\langle v, \pi, n\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto} \langle \pi, v, n\rangle_{\mathsf{ct}} \qquad \text{(Switch)}$$

$$\langle \bullet\ t :: \pi, v, n\rangle_{\mathsf{ct}} \overset{\tau}{\mapsto} \langle t, v\ \bullet :: \pi, n\rangle_{\mathsf{ev}} \qquad \text{(Arg)}$$

$$\langle \lambda x.t\ \bullet :: \pi, v, n\rangle_{\mathsf{ct}} \overset{\tau}{\mapsto} \langle [v\,/\,x]t, \pi, n\rangle_{\mathsf{ev}} \qquad \text{(Beta)}$$

$$\langle [], v, n\rangle_{\mathsf{ct}} \overset{\circledcirc}{\mapsto} \langle v\ \bullet :: [], n, n+1\rangle_{\mathsf{ct}} \qquad \text{(Val)}$$

$$\langle f\ \bullet :: \pi, v, n\rangle_{\mathsf{ct}} \overset{f}{\mapsto}\!\!\blacktriangledown \langle v\ \bullet :: [], n, n+1\rangle_{\mathsf{ct}} \qquad \text{(Stuck-Val)}$$

$$\langle f\ \bullet :: \pi, v, n\rangle_{\mathsf{ct}} \overset{f}{\mapsto}\!\!\blacktriangleright \langle \pi, n, n+1\rangle_{\mathsf{ct}} \qquad \text{(Stuck-Context)}$$

Figure 8: NFB machine for call by value

transformation [Las05]. It is not complete w.r.t. the contextual equivalence of the call-by-value λ-calculus [Las05, Example 3.2].

We internalize applicative bisimilarity as in call-by-name (Section 5), by considering a λ-calculus with de Bruijn indices and an environment-based CK machine, i.e., the CEK abstract machine [FF86, FFF09]. The definition of the bisimilarity differs from call-by-name only in the fact that the testing argument must be a λ-abstraction. Consequently, we just use the argument-generating rules from Figure 4 on top of the CEK machine, and change the rule Restart so that it can trigger only if the constructed term is a λ-abstraction. Internalizing contextual equivalence in call-by-value simply consists in adding the unmodified rules for generating a context (Section 5.4) on top of the CEK machine. In the rest of this section, we discuss only the more interesting case of the NFB machine and its translation in HOcore.

6.3. **NFB Machine.** We extend the CK machine into a NFB machine in Figure 8. Configurations include a counter n to generate fresh variables, as in call-by-name. The first four machine steps correspond to the CK machine, while the last three ones compare normal forms.

If we get a value, we flag $\circledcirc$ and apply it to a fresh variable (transition Val). In the case of a stuck term $\langle f\ \bullet :: \pi, v\rangle_{\mathsf{ct}}$, we have to choose whether we test v (transition Stuck-Val) or π (transition Stuck-Context). In both cases, we flag the free variable f. The test for values is the same as in the transition Val. If we choose the stack π, we simply restart the machine in continuation mode with π and a fresh variable.

We prove that the resulting machine equivalence coincides with normal-form bisimilarity using the same proof technique as in call-by-name.

Theorem 6.3. *$t \approx_{\mathsf{nf}} s$ iff there exists $n > \max(\mathsf{fv}(t) \cup \mathsf{fv}(s))$ such that $\langle t, [], n\rangle_{\mathsf{ev}} \approx_{\mathsf{m}} \langle s, [], n\rangle_{\mathsf{ev}}$.*

Remark 6.4. In extensions of the λ-calculus where functions are not the only values (e.g., with booleans or integers as primitive constructs), a value is no longer equivalent to its η-expansion in general, and we would have to consider the finer normal-form bisimilarity which distinguishes values based on their kind. To internalize it in the plain λ-calculus, we

would replace Val with the transitions Done and Lambda below, and Stuck-Val with the new step below.

$$\langle [], f, n\rangle_{\mathsf{ct}} \overset{f}{\mapsto}\overset{\star}{\mapsto} \tag{Done}$$

$$\langle [], \lambda x.t, n\rangle_{\mathsf{ct}} \overset{\circledcirc}{\mapsto} \langle [n \,/\, x]t, [], n+1\rangle_{\mathsf{ev}} \tag{Lambda}$$

$$\langle f\ \bullet :: \pi, v, n\rangle_{\mathsf{ct}} \overset{f}{\mapsto}\overset{\blacktriangledown}{\mapsto} \langle [], v, n\rangle_{\mathsf{ct}} \tag{Stuck-Val}$$

If we get a free variable, we signal it and we are done. We instantiate λ-abstractions with a fresh variable. Finally, the transition Stuck-Val goes to $\langle [], v, n\rangle_{\mathsf{ct}}$, after which we immediately apply either Done or Lambda depending on v.

We present the encoding of the call-by-value NFB machine in Figure 9. We follow the same principles as in Section 4.3. For example, we still use an output on b to describe what to do when the stack is empty—run the $\mathsf{Restart}$ process; the message on b is simply discarded when the stack is not empty. The $\mathsf{Restart}$ process flags $\circledcirc$ and then recreates the configuration where the value in *val* is applied to a fresh variable, represented by the counter n stored in k.

A free variable f is observable in a stuck configuration $\langle f\ \bullet :: \pi, v, n\rangle_{\mathsf{ct}}$; in that case, it signals itself with a sequence of inputs on *suc* ended by an input on z, and then it non-deterministically chooses between testing v or π. The $\mathsf{Chce}(P_v, P_\pi, P_n)$ process is making that choice, applying P_n to P_v in the first case, or running P_n in P_π in the second one. Unlike in call-by-name, we do not have to go recursively through the stack: as a result, although the CK machine is more complex than the KAM, the translation of the call-by-value NFB machine is arguably simpler than the call-by-name one.

We can prove full abstraction between the call-by-value NFB machines and their translated HOcore processes as in Section 4.4, from which we can deduce full abstraction between HOcore and the λ-calculus with normal-form bisimilarity.

Theorem 6.5. *$t \approx_{\mathsf{nf}} s$ iff there exists $n > \max(\mathsf{fv}(t) \cup \mathsf{fv}(s))$ such that $[\![\langle t, [], n\rangle_{\mathsf{ev}}]\!] \approx_{\mathsf{HO}} [\![\langle s, [], n\rangle_{\mathsf{ev}}]\!]$.*

7. Control Operators

The benefit of using abstract machines as an intermediary step is that our encoding can easily be extended to control operators, the semantics of which can be expressed using abstract machines. Among the existing control operators, we choose to encode the $\lambda\mu$-calculus [Par92], to show how to deal with a calculus with multiple binders.

7.1. Extended KAM for the Call-by-Name $\lambda\mu$-calculus.

The $\lambda\mu$-calculus extends the λ-calculus with *names* or *continuation variables*—ranged over by α and β—which represent an unknown continuation, and a μ operator to capture continuations. The syntax of terms becomes as follows.

$$t, s ::= x \mid \lambda x.t \mid t\ s \mid \mu\alpha.[\beta]t$$

The semantics of the $\lambda\mu$-calculus is not defined on plain terms but on *named terms* of the form $[\alpha]t$. In $\mu\alpha.[\beta]t$, the occurrences of α in $[\beta]t$ are bound. We write $\mathsf{fn}(t)$ and $\mathsf{fn}([\alpha]t)$ for the set of free names of a term or named term. A term is closed if it has no free variable

$$
\begin{aligned}
[\![t\ s]\!] &\triangleq c(p).([\![t]\!] \parallel \overline{c}\langle \mathsf{Arg}([\![s]\!], p)\rangle) \\
[\![v]\!] &\triangleq c(p).(p \parallel \overline{val}\langle [\![v]\!]_{\mathsf{v}}\rangle \parallel \overline{b}\langle \mathsf{Restart}\rangle) \\
[\![\lambda x.t]\!]_{\mathsf{v}} &\triangleq val(x).b(\_).[\![t]\!] \\
[\![f]\!]_{\mathsf{v}} &\triangleq [\![f]\!]_{\mathsf{Int}}
\end{aligned}
$$

$$
\begin{aligned}
[\![0]\!]_{\mathsf{Int}} &\triangleq z(\_).val(x).c(p).k(y).b(\_).\mathsf{Chce}(x,p,y) \\
[\![n+1]\!]_{\mathsf{Int}} &\triangleq suc(\_).[\![n]\!]_{\mathsf{Int}} \\
\mathsf{Restart} &\triangleq \circledcirc(\_).val(x).k(y).(\mathsf{Fun}(x,[\![]]\!]) \parallel \overline{val}\langle y\rangle \parallel \overline{k}\langle suc(\_).y\rangle \parallel \overline{b}\langle \mathbf{0}\rangle) \\
\mathsf{Chce}(P_v,P_\pi,P_n) &\triangleq \blacktriangledown(\_).(\mathsf{Fun}(P_v,[\![]]\!]) \parallel \overline{val}\langle P_n\rangle \parallel \overline{k}\langle suc(\_).P_n\rangle \parallel \overline{b}\langle \mathsf{Restart}\rangle) \\
&\quad + \blacktriangleright(\_).(P_\pi \parallel \overline{val}\langle P_n\rangle \parallel \overline{k}\langle suc(\_).P_n\rangle \parallel \overline{b}\langle \mathsf{Restart}\rangle)
\end{aligned}
$$

$$
\begin{aligned}
[\![]]\!] &\triangleq b(x).x \\
[\![\bullet\ t :: \pi]\!] &\triangleq \mathsf{Arg}([\![t]\!],[\![\pi]\!]) \\
[\![v\ \bullet :: \pi]\!] &\triangleq \mathsf{Fun}([\![v]\!]_{\mathsf{v}},[\![\pi]\!]) \\
\mathsf{Arg}(P_t,P_\pi) &\triangleq val(x).b(\_).(P_t \parallel \overline{c}\langle \mathsf{Fun}(x,P_\pi)\rangle) \\
\mathsf{Fun}(P_v,P_\pi) &\triangleq P_v \parallel \overline{c}\langle P_\pi\rangle
\end{aligned}
$$

$$
\begin{aligned}
[\![\langle t,\pi,n\rangle_{\mathsf{ev}}]\!] &\triangleq [\![t]\!] \parallel \overline{c}\langle [\![\pi]\!]\rangle \parallel \overline{k}\langle [\![n]\!]_{\mathsf{Int}}\rangle \\
[\![\langle \pi,v,n\rangle_{\mathsf{ct}}]\!] &\triangleq [\![\pi]\!] \parallel \overline{val}\langle [\![v]\!]_{\mathsf{v}}\rangle \parallel \overline{k}\langle [\![n]\!]_{\mathsf{Int}}\rangle \parallel \overline{b}\langle \mathsf{Restart}\rangle
\end{aligned}
$$

Figure 9: Encoding of the call-by-value NFB machine into HOcore

and no free name. A named term $[\alpha]t$ cannot be closed as it has at least α among its free names.

We define the semantics of the $\lambda\mu$-calculus using an extension of the KAM, meaning that the continuations captured by the μ operator are represented by stacks. Like in the call-by-name λ-calculus, a stack represents a sequence of applications of terms; the difference is that it is ended by a name which represents the toplevel in which the term is executed.

$$\pi ::= t :: \pi \mid \alpha$$

The term $\mu\alpha.[\beta]t$ captures such a stack π to replace α in $[\beta]t$ with it. *Structural substitution* [AH08] $\langle \pi \,/\, \alpha\rangle t$ produces a plain term where the name α is replaced by π in t. It is defined in Figure 10, alongside two auxiliary operations: name substitution $\langle \pi \,/\, \alpha\rangle\alpha$ which produces a stack, and plugging $\pi\{t\}$ which produces a named term. Plugging simply reconstructs around t the applications represented by the stack π. In the case $\langle \pi/\alpha\rangle\mu\beta_1.[\beta_2]t$, we assume the free name α to be distinct from β_1, which is always possible using α-conversion. The captured stack π is restored when $\beta_2 = \alpha$, as we plug $\langle \pi \,/\, \alpha\rangle t$ inside π.

$$\langle \pi \,/\, \alpha \rangle \alpha \triangleq \pi \qquad \alpha\{t\} \triangleq [\alpha]t$$
$$\langle \pi \,/\, \alpha \rangle \beta \triangleq \beta \text{ if } \alpha \neq \beta \qquad (s :: \pi)\{t\} \triangleq \pi\{t\ s\}$$

$$\begin{aligned}
\langle \pi \,/\, \alpha \rangle x &\triangleq x \\
\langle \pi \,/\, \alpha \rangle \lambda x.t &\triangleq \lambda x.\langle \pi \,/\, \alpha \rangle t \\
\langle \pi \,/\, \alpha \rangle (t\ s) &\triangleq \langle \pi \,/\, \alpha \rangle t\ \langle \pi \,/\, \alpha \rangle s \\
\langle \pi \,/\, \alpha \rangle \mu\beta_1.[\beta_2]t &\triangleq \mu\beta_1.(\langle \pi \,/\, \alpha \rangle \beta_2)\{\langle \pi \,/\, \alpha \rangle t\}
\end{aligned}$$

Figure 10: Structural Substitution

The semantics is given by the extended KAM, defined on the same configurations as in Section 3.

$$C ::= \langle t, \pi \rangle \qquad \text{(configurations)}$$

$$\begin{aligned}
\langle t\ s, \pi \rangle &\mapsto \langle t, s :: \pi \rangle && \text{(Push)} \\
\langle \lambda x.t, s :: \pi \rangle &\mapsto \langle [s \,/\, x]t, \pi \rangle && \text{(Grab)} \\
\langle \mu\alpha.[\beta]t, \pi \rangle &\mapsto \langle \langle \pi \,/\, \alpha \rangle t, \langle \pi \,/\, \alpha \rangle \beta \rangle && \text{(Capture)}
\end{aligned}$$

The Push and Grab steps are unchanged. The last step captures the stack π and replaces α with it, thanks to the operations defined in Figure 10. To evaluate a closed term, we assume the existence of a distinguished name tp representing the toplevel [AH08], and we execute $\langle t, \mathsf{tp} \rangle$.

Example 7.1. Unlike the μ operator, the control operator call/cc from Scheme leaves the captured stack in place when it triggers. This behaviour can be encoded in the $\lambda\mu$-calculus as follows [Par92]:

$$\mathsf{callcc} \triangleq \lambda x.\mu\alpha.[\alpha]x\ \lambda y.\mu\beta.[\alpha]y$$

The captured stack is immediately restored, but also saved in the value passed to x. To illustrate the extended KAM, we run it on an example. Let $id \triangleq \lambda x.x$, $\Omega \triangleq (\lambda x.x\ x)\ (\lambda x.x\ x)$, $v \triangleq \lambda z.z\ id\ \Omega$, and t be any term. We compute $\mathsf{callcc}\ v\ t$ in the name tp.

$$\begin{aligned}
\langle \mathsf{callcc}\ v\ t, \mathsf{tp} \rangle &\mapsto^2 \langle \mathsf{callcc}, v :: t :: \mathsf{tp} \rangle && \text{(Push - Push)} \\
&\mapsto \langle \mu\alpha.[\alpha]v\ \lambda y.\mu\beta.[\alpha]y, t :: \mathsf{tp} \rangle && \text{(grab)} \\
&\mapsto \langle v\ \lambda y.\mu\beta.[\mathsf{tp}]y\ t, t :: \mathsf{tp} \rangle && \text{(Capture)} \\
&\mapsto^2 \langle (\lambda y.\mu\beta.[\mathsf{tp}]y\ t)\ id\ \Omega, t :: \mathsf{tp} \rangle && \text{(Push - Grab)} \\
&\mapsto^3 \langle \mu\beta.[\mathsf{tp}]id\ t, \Omega :: t :: \mathsf{tp} \rangle && \text{(Push - Push - Grab)} \\
&\mapsto \langle id\ t, \mathsf{tp} \rangle && \text{(Capture)} \\
&\mapsto^2 \langle t, \mathsf{tp} \rangle && \text{(Push - Grab)}
\end{aligned}$$

After the first capture, the stack $t :: \mathsf{tp}$ is still in place, while the stack $\Omega :: t :: \mathsf{tp}$ disappears during the second capture, because β does not occur in the body of the μ operator.

Using a substitution-based semantics for the μ operator means it can easily be encoded in HOcore as a higher-order communication. We translate names as HOcore variables, chosen to be distinct from the set of translated term variables. When we evaluate closed terms, the distinguished name tp is the end of the stack and is translated as such. The μ operator is translated as follows.

$$[\![\mathsf{tp}]\!] \triangleq \overline{b}\langle \mathbf{0} \rangle$$

$$[\![\mu\alpha.[\beta]t]\!] \triangleq c(\alpha).([\![t]\!] \parallel \overline{c}\langle\beta\rangle)$$

The rest of the translation is the same as in Section 3. The translated μ operator reflects the Capture step by receiving the current stack on c. Because we execute closed terms, either $\beta = \alpha$, or β is bound higher in the term and has been substituted before we get to this point. The term $[\![t]\!]$ is therefore executed in a previously captured stack.

The correspondence between the extended KAM and its HOcore translation can be stated the same way as for the regular KAM (Theorem 3.1). The Capture step corresponds to exactly one communication in the translation of the machine.

7.2. **NFB Machine.** To internalize normal-form bisimilarity into the extended KAM, we need to distinguish not only free variables f from bound variables x, but also free names ϕ, ψ from bound names δ. Indeed, normal-form bisimilarity uses fresh names to compare terms and discriminates terms based on these names. We change the syntax of terms as follows.

$$\alpha ::= \phi \mid \delta \qquad t, s ::= f \mid x \mid \lambda x.t \mid t\ s \mid \mu\delta.[\alpha]t$$

We assume terms to be well-formed, i.e., to not contain a bound variable or bound name without its binder. The name at the end of a stack is necessarily free.

$$\pi ::= t :: \pi \mid \phi$$

Given two stacks π_1, π_2, we write $\pi_1 \mathcal{R} \pi_2$ if their elements are pairwise related and they are ended by the same name. Formally, we have either $\pi_1 = \pi_2 = \phi$ for some ϕ, or $\pi_1 = t :: \pi_1'$, $\pi_2 = s :: \pi_2'$, $t \mathcal{R} s$, and $\pi_1' \mathcal{R} \pi_2'$. The definition of normal-form bisimilarity for the call-by-name $\lambda\mu$-calculus is as follows.

Definition 7.2. A symmetric relation $\mathcal{R}$ is a normal-form bisimulation if $t \mathcal{R} s$ and ϕ fresh implies:

- if $\langle t, \phi\rangle \mapsto^* \langle \lambda x.t', \psi\rangle$, then there exists s' such that $\langle s, \phi\rangle \mapsto^* \langle \lambda x.s', \psi\rangle$ and $[f\,/\,x]t' \mathcal{R} [f\,/\,x]s'$ for a fresh f;
- if $\langle t, \phi\rangle \mapsto^* \langle f, \pi\rangle$, then there exists π' such that $\langle s, \phi\rangle \mapsto^* \langle f, \pi'\rangle$ and $\pi \mathcal{R} \pi'$.

Normal-form bisimilarity is the largest normal-form bisimulation.

Normal-form bisimilarity is not complete w.r.t. contextual equivalence of the $\lambda\mu$-calculus when defined either with weak-head reduction [Las99a] or head reduction [Las06]. For an extended $\lambda\mu$-calculus— the so-called $\Lambda\mu$ calculus [Sau05]—with head reduction, it coincides with solvable equivalence [Bar84, Las06].

The bisimulation definition is almost the same as in λ-calculus (Definition 4.1) except for names. Terms are compared within some fresh name ϕ for each bisimulation round, as using a single name like tp for all rounds would not be discriminating enough. It would relate for instance x and $\mu\delta.[\mathsf{tp}]x$, because $\langle x, \mathsf{tp}\rangle$ and $\langle \mu\delta.[\mathsf{tp}]x, \mathsf{tp}\rangle$ evaluate to $\langle x, \mathsf{tp}\rangle$, but they behave differently: the second term discards its continuation while the first one does not. Similarly, we expect the resulting normal forms to exhibit a common name ψ (not

$$\langle t\ s, \pi, n, m\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto} \langle t, s :: \pi, n, m\rangle_{\mathsf{ev}} \qquad \text{(Push)}$$

$$\langle \lambda x.t, s :: \pi, n, m\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto} \langle [s \,/\, x]t, \pi, n, m\rangle_{\mathsf{ev}} \qquad \text{(Grab)}$$

$$\langle \mu\delta.[\alpha]t, \pi, n, m\rangle_{\mathsf{ev}} \overset{\tau}{\mapsto} \langle \langle \pi \,/\, \delta\rangle t, \langle \pi \,/\, \delta\rangle \alpha, n, m\rangle_{\mathsf{ev}} \qquad \text{(Capture)}$$

$$\langle \lambda x.t, \phi, n, m\rangle_{\mathsf{ev}} \overset{\phi}{\mapsto}\!\overset{\circledcirc}{\mapsto} \langle [n \,/\, x]t, m, n+1, m+1\rangle_{\mathsf{ev}} \qquad \text{(Lambda)}$$

$$\langle f, \pi, n, m\rangle_{\mathsf{ev}} \overset{f}{\mapsto} \langle \pi, n, m\rangle_{\mathsf{ct}} \qquad \text{(Var)}$$

$$\langle \phi, n, m\rangle_{\mathsf{ct}} \overset{\phi}{\mapsto}\!\overset{\star}{\mapsto} \qquad \text{(Done)}$$

$$\langle t :: \pi, n, m\rangle_{\mathsf{ct}} \overset{\blacktriangledown}{\mapsto} \langle t, m, n, m+1\rangle_{\mathsf{ev}} \qquad \text{(Enter)}$$

$$\langle t :: \pi, n, m\rangle_{\mathsf{ct}} \overset{\blacktriangleright}{\mapsto} \langle \pi, n, m\rangle_{\mathsf{ct}} \qquad \text{(Skip)}$$

Figure 11: NFB machine for the call-by-name $\lambda\mu$-calculus

necessarily equal to ϕ). Terms evaluating to normal forms with distinct toplevel names would imply that they return to different continuations, and a surrounding context could easily distinguish them by picking a terminating continuation for a term and a diverging continuation for the other.

We adapt the λ-calculus NFB machine to take these changes into account, resulting in the machine of Figure 11. In the machine, free names are natural numbers, and configurations $\langle t, \pi, n, m\rangle_{\mathsf{ev}}$ include an extra counter m to generate fresh names. This counter is used as a new stack whenever we restart a computation, in the Lambda and Enter steps. The Lambda and Done steps now flag the free name f of the resulting normal forms. The rest of the machine is the same as in λ-calculus, except for the Capture step.

The translation of the NFB machine for the $\lambda\mu$-calculus into HOcore follows the same principles as in λ-calculus. We just need to be careful to distinguish between numbers representing free variables from those standing for free names—by using distinct HOcore names suc_{v}, z_{v} and suc_{n}, z_{n}. We can prove the equivalence between normal-form bisimilarity and its machine, and between the NFB machine and its translation as in Section 4.

Theorem 7.3. *$t \approx_{\mathsf{nf}} s$ iff there exists $n > \max(\mathsf{fv}(t) \cup \mathsf{fv}(s))$ and $m > \max(\mathsf{fn}(t) \cup \mathsf{fn}(s))$ such that $[\![\langle t, m, n, m+1\rangle_{\mathsf{ev}}]\!] \approx_{\mathsf{HO}} [\![\langle s, m, n, m+1\rangle_{\mathsf{ev}}]\!]$.*

7.3. **Environment-based Abstract Machine.** As in λ-calculus (Section 5), internalizing applicative bisimilarity requires to generate a testing argument. Because we cannot generate binders in the HOcore translation, we consider a de Bruijn syntax for terms and an environment-based machine. However, unlike in λ-calculus, named terms are not completely closed, because of the toplevel name, and when we play the bisimulation game, considering a single free name tp is not enough, as it would not be sound, like with normal-form bisimilarity (cf Section 7.2). We therefore need to consider three distinct families of de Bruijn indices: term variables n, bound names δ, and free names ϕ. The de Bruijn syntax of the $\lambda\mu$-calculus is therefore as follows.

$$\alpha ::= \delta \mid \phi \qquad t, s ::= n \mid \lambda.t \mid t\ s \mid \mu.[\alpha]t$$

$$
\begin{array}{rr}
e ::= \eta :: e \mid \epsilon_{\mathsf{t}} & \text{(term-variables environments)} \\
\Delta ::= \pi :: \Delta \mid \epsilon_{\mathsf{b}} & \text{(bound-names environments)} \\
\eta ::= (t, e, \Delta) & \text{(closures)} \\
\pi ::= \eta :: \pi \mid \alpha & \text{(stacks)} \\
C ::= \langle t, e, \Delta, \pi \rangle_{\mathsf{ev}} \mid \langle \delta, \Delta, \eta \rangle_{\mathsf{bn}} & \text{(configurations)}
\end{array}
$$

$$
\begin{array}{rr}
\langle t\ s, e, \Delta, \pi \rangle_{\mathsf{ev}} \xmapsto{\tau} \langle t, e, \Delta, (s, e, \Delta) :: \pi \rangle_{\mathsf{ev}} & \text{(Push)} \\
\langle 0, (t, e, \Delta) :: d, \Gamma, \pi \rangle_{\mathsf{ev}} \xmapsto{\tau} \langle t, e, \Delta, \pi \rangle_{\mathsf{ev}} & \text{(Zero-Term)} \\
\langle n+1, \eta :: e, \Delta, \pi \rangle_{\mathsf{ev}} \xmapsto{\tau} \langle n, e, \Delta, \pi \rangle_{\mathsf{ev}} & \text{(Env-Term)} \\
\langle \lambda.t, e, \Delta, \eta :: \pi \rangle_{\mathsf{ev}} \xmapsto{\tau} \langle t, \eta :: e, \Delta, \pi \rangle_{\mathsf{ev}} & \text{(Grab)} \\
\langle \mu.[\alpha]t, e, \Delta, \pi \rangle_{\mathsf{ev}} \xmapsto{\tau} \langle t, e, \pi :: \Delta, \alpha \rangle_{\mathsf{ev}} & \text{(Capture)} \\
\langle \lambda.t, e, \Delta, \delta \rangle_{\mathsf{ev}} \xmapsto{\tau} \langle \delta, \Delta, (\lambda.t, e, \Delta) \rangle_{\mathsf{bn}} & \text{(Restore)}
\end{array}
$$

$$
\begin{array}{rr}
\langle 0, \pi :: \Gamma, (t, e, \Delta) \rangle_{\mathsf{bn}} \xmapsto{\tau} \langle t, e, \Delta, \pi \rangle_{\mathsf{ev}} & \text{(Zero-BName)} \\
\langle \delta + 1, \pi :: \Delta, \eta \rangle_{\mathsf{bn}} \xmapsto{\tau} \langle \delta, \Delta, \eta \rangle_{\mathsf{bn}} & \text{(Env-BName)}
\end{array}
$$

Figure 12: Environment-based KAM for the $\lambda\mu$-calculus

We present in Figure 12 the environment-based extended KAM. It is similar to the Streicher and Reus machine [SR98], except their syntax uses names and not de Bruijn indices, and they use a single environment for term variables and bound names. Instead, we distinguish the environment mapping term variables to closures (ranged by e, d) from the one mapping bound names to stacks (ranged by Δ, Γ). An extra environment mapping free names to stacks is added when we internalize applicative bisimilarity (Section 7.4). Closures η are triples (t, e, Δ), and stacks π are composed of closures and ended by a free or bound name.

The first four steps are as in the λ-calculus. The Capture step replaces the current stack π with α and puts π on top of the bound-name environment Δ. Whenever a λ-abstraction is executed within a stack composed only of a bound name δ, we restore the previously captured stack, by looking into Δ. It is the role of the Zero-BName and Env-BName steps, which operate on dedicated configurations of the form $\langle \delta, \Delta, \eta \rangle_{\mathsf{bn}}$.

7.4. **AB machine.** The definition of the sound and complete applicative bisimilarity for the $\lambda\mu$-calculus [BL14] is more intricate than in λ-calculus, because we compare named terms. We first recall its definition [BL14] and using named binders, before moving to de Bruijn indices, and then to an AB machine.

7.4.1. *Informal definition with named binders.* Assume two named terms $[\alpha]t$ and $[\alpha]s$ where t and s are closed. If they evaluate to respectively $[\alpha]\lambda x.t'$ and $[\alpha]\lambda x.s'$, a surrounding context cannot test them by applying them to an arbitrary closed term t'', as $([\alpha]\lambda x.t')\ t''$ is not a valid $\lambda\mu$-term. The simplest named term that can be built to lead to such an application is $[\beta](\mu\alpha.[\alpha]\lambda x.t')\ t''$ for a fresh β. Indeed, the μ-binder captures the named context $[\beta]\,\Box\ t''$ (represented as a stack $t'' :: \beta$) to reduce to $[\beta](\lambda x.\langle t'' :: \beta \,/\, \alpha\rangle t')\ t''$; the structural substitution $\langle t'' :: \beta \,/\, \alpha\rangle t'$ is necessary because α may occur in t'. The β-reduction is now possible and the previous named term reduces to $[\beta][t'' \,/\, x]\langle t'' :: \beta \,/\, \alpha\rangle t'$. We get the same sequence of reductions with $[\alpha]s$, so that applicative bisimilarity compares

$$[\beta][t'' \,/\, x]\langle t'' :: \beta \,/\, \alpha\rangle t' \text{ and } [\beta][t'' \,/\, x]\langle t'' :: \beta \,/\, \alpha\rangle s'.$$

We need to write such a test with de Bruijn indices and environments. We represent the successive fresh names α, β, ... used by the bisimilarity with indices 0, 1, ... Assume we evaluate the closed closures (t, e, Δ) and (s, d, Γ) in the free name 0, resulting respectively into $(\lambda.t', e', \Delta')$ and $(\lambda.s', d', \Gamma')$ in the same name 0. Let t'' be a closed term. The term substitution $[t'' \,/\, x]\cdot$ is represented as in plain λ-calculus, by extending e' and d' with the closure corresponding to t''. To represent the structural substitution $\langle t'' :: \beta \,/\, \alpha\rangle\cdot$ we introduce an extra environment dedicated to free names.

7.4.2. *Testing environments.* A *free-names environment* Φ is a mapping from free names (represented as de Bruijn indices) to stacks, defined as follows.

$$\Phi ::= \pi :: \Phi \mid \epsilon_{\mathsf{f}} \qquad \text{(free-names environments)}$$

This environment is extended by the bisimilarity each time it tests pairs of named values. We write $\ulcorner t'' \urcorner$ for the closure $(t'', \epsilon_{\mathsf{t}}, \epsilon_{\mathsf{b}}, \epsilon_{\mathsf{f}})$ used in these tests. The stack $t'' :: \beta$ is represented as $\ulcorner t'' \urcorner :: 1$, and Φ maps 0 to $\ulcorner t'' \urcorner :: 1$. A subsequent test would extend Φ to map 1 to $\ulcorner t''' \urcorner :: 2$ for some t''', then a following test would map 2 to another closure, *etc.*

We see that the bisimilarity extends Φ at the end, and we write $\Phi ::: \pi$ for such an extension. We also notice that the resulting Φ maps each free name to a stack containing only one element. More precisely, an environment Φ generated by successive bisimilarity tests is either empty, or its i^{th} element is of the form $\ulcorner t \urcorner :: i$ for some closed term t. We say such an environment is a *testing environment*, of rank ϕ if it has ϕ elements. A testing environment of rank $\phi > 0$ has only ϕ as free-name index without a mapping, at the end of its last stack. For example, a testing environment of rank 2 is of the form $\pi_0 :: \pi_1 :: \epsilon_{\mathsf{f}}$, mapping 0 to $\pi_0 = \ulcorner t_0 \urcorner :: 1$ for some closed t_0, and 1 to $\pi_1 = \ulcorner t_1 \urcorner :: 2$ for some closed t_1.

7.4.3. *Applicative bisimilarity.* We extend closures to include testing environments $\eta ::= (t, e, \Delta, \Phi)$, and we say that η is of rank ϕ when its testing environment is of rank ϕ. A stack is closed if it is composed only of closed closures. A closure is closed if e contains more elements than the highest term-variable index in t, Δ contains more elements than the highest bound-name index in t, e contains only closed closures, and Δ only closed stacks.

We add the testing environment Φ to configurations, and we define steps to look in Φ for the stack corresponding to a given free-name index. We discuss these changes in details when defining the AB machine. We adapt the definition of applicative bisimilarity [BL14] to our setting as follows.

Definition 7.4. A symmetric relation $\mathcal{R}$ on closed closures is an applicative bisimulation if $(t, e, \Delta, \Phi)\ \mathcal{R}\ (s, d, \Gamma, \Phi)$ with Φ of rank ϕ and $\langle t, e, \Delta, \Phi, \phi\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*}\ \langle \lambda.t', e', \Delta', \Phi, \phi\rangle_{\mathsf{ev}}$ implies that there exists s', d', and Γ' such that $\langle s, d, \Gamma, \Phi, \phi\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto}{}^{*}\ \langle \lambda.s', d', \Gamma', \Phi, \phi\rangle_{\mathsf{ev}}$ and for all closed t'', we have

$$(t', \ulcorner t''\urcorner :: e', \Delta', \Phi ::: (\ulcorner t''\urcorner :: \phi + 1))\ \mathcal{R}\ (s', \ulcorner t''\urcorner :: d', \Gamma', \Phi ::: (\ulcorner t''\urcorner :: \phi + 1)).$$

Applicative bisimilarity $\approx_{\mathsf{app}}$ is the largest applicative bisimulation.

Applicative bisimilarity executes closed closures with the same testing environment of rank ϕ within the stack composed only of the free name ϕ. It then compares the resulting values by considering a closed closure $\ulcorner t''\urcorner$ to instantiate the λ-abstractions, and to extend the testing environment which becomes of rank $\phi + 1$.

7.4.4. *AB machine.* We present the most interesting steps of the corresponding AB machine in Figure 13. Evaluation configurations $\langle t, e, \Delta, \Phi, \pi\rangle_{\mathsf{ev}}$ now include the testing environment Φ, which is passed along unchanged in the steps corresponding to the extended KAM (omitted in Figure 13).

Whenever a λ-abstraction is executed in a stack composed only of a free name ϕ, we have two possibilities: either ϕ is mapped in Φ, otherwise we have to generate the closure $\ulcorner t\urcorner$ of the bisimulation test. The configuration $\langle \phi, \Phi, \eta, \Phi_r, \phi'\rangle_{\mathsf{fn}}$ is designed with these two cases in mind. Its main role is to look for ϕ in Φ and to restore the corresponding stack if the mapping exists.

If ϕ is not mapped, then it is stored as the last parameter of the fn mode in the Lookup-FName step, and passed along during the argument generation process. Its successor is needed to create the stack $\ulcorner t\urcorner :: \phi + 1$ (Init-Rev step).

The second-to-last parameter Φ_r of the fn mode is used to compute the reverse of Φ. Indeed, we remind that applicative bisimilarity extends Φ from its tail. To do so, we reverse Φ, add the generated closure, and reverse the result again. We set Φ_r to empty in the Lookup-FName step, and then the elements of Φ are pushed on Φ_r when we go through Φ in the Env-FName step. As a result, when Φ is empty in the Arg transition, Φ_r contains the original environment in reverse.

The Arg step initiates the argument generation. It triggers when Φ is empty, and we know that the index at this point is 0, because the only free-name index without a mapping allowed by design is the rank of Φ, i.e., its size. As in Section 5.2, we start argument generation with the leftmost de Bruijn index representing a term variable, and then generate the term from left to right. Compared to λ-calculus, the ind and tm configurations carry three extra parameters: the already discussed Φ_r and ϕ, and κ_μ, which counts how many μ-binders are necessary for the generated term to be closed. In Figure 13, we omit the steps that are the same as in Figure 4 up to these three parameters.

We generate a term $\mu.[\delta]t$ by starting with the bound name index δ (Ind-Mu and Suc-Mu steps). We then add the μ-binder and update the κ_μ counter accordingly (Mu step).

After generating the argument t, we step to a configuration of the form $\langle \Phi_r, \Phi, \eta, \pi\rangle_{\mathsf{envfn}}$, the goal of which is to construct the extended testing environment. The Init-Rev step constructs the bottom of the extended environment and remembers $\ulcorner t\urcorner :: \phi + 1$ as the new stack. The Rev step goes through Φ_r and pushes its elements on Φ to reconstruct the original environment on top of its new bottom. When Φ_r is empty, the Restart step

$$\langle t\ s, e, \Delta, \Phi, \pi\rangle_{\mathsf{ev}} \xrightarrow{\tau} \langle t, e, \Delta, \Phi, (s, e, \Delta, \Phi) :: \pi\rangle_{\mathsf{ev}} \qquad \text{(Push)}$$

$$\vdots \qquad \text{(cf. Figure 12)}$$

$$\langle \lambda.t, e, \Delta, \Phi, \phi\rangle_{\mathsf{ev}} \xrightarrow{\tau} \langle \phi, \Phi, (\lambda.t, e, \Delta, \Phi), \epsilon_{\mathsf{f}}, \phi\rangle_{\mathsf{fn}} \qquad \text{(Lookup-FName)}$$

$$\langle 0, \pi :: \Phi', (t, e, \Delta, \Phi), \Phi_r, \phi\rangle_{\mathsf{fn}} \xrightarrow{\tau} \langle t, e, \Delta, \Phi, \pi\rangle_{\mathsf{ev}} \qquad \text{(Zero-FName)}$$

$$\langle \phi + 1, \pi :: \Phi, \eta, \Phi_r, \phi'\rangle_{\mathsf{fn}} \xrightarrow{\tau} \langle \phi, \Phi, \eta, \pi :: \Phi_r, \phi'\rangle_{\mathsf{fn}} \qquad \text{(Env-FName)}$$

$$\langle 0, \epsilon_{\mathsf{f}}, \eta, \Phi_r, \phi\rangle_{\mathsf{fn}} \xrightarrow{\circledcirc} \langle 0, 1, 0, \odot, \eta, \Phi_r, \phi\rangle_{\mathsf{ind}} \qquad \text{(Arg)}$$

$$\langle n, \kappa, \kappa_\mu, \rho, \eta, \Phi_r, \phi\rangle_{\mathsf{ind}} \xrightarrow{\boxplus} \langle n + 1, \kappa + 1, \kappa_\mu, \rho, \eta, \Phi_r, \phi\rangle_{\mathsf{ind}} \qquad \text{(Suc)}$$

$$\vdots \qquad \text{(cf. Figure 4)}$$

$$\langle t, \kappa, \kappa_\mu, \rho, \eta, \Phi_r, \phi\rangle_{\mathsf{tm}} \xrightarrow{i\mu} \langle 0, t, \kappa, \kappa_\mu, \rho, \eta, \Phi_r, \phi\rangle_{\mathsf{indMu}} \qquad \text{(Ind-Mu)}$$

$$\langle \delta, t, \kappa, \kappa_\mu, \rho, \eta, \Phi_r, \phi\rangle_{\mathsf{indMu}} \xrightarrow{s\mu} \langle \delta + 1, t, \kappa, \kappa_\mu, \rho, \eta, \Phi_r, \phi\rangle_{\mathsf{indMu}} \qquad \text{(Suc-Mu)}$$

$$\langle \delta, t, \kappa, \kappa_\mu, \rho, \eta, \Phi_r, \phi\rangle_{\mathsf{indMu}} \xrightarrow{\mu} \langle \mu.[\delta]t, \kappa, \mathsf{max}(\delta + 1, \kappa_\mu) - 1, \rho, \eta, \Phi_r, \phi\rangle_{\mathsf{tm}} \qquad \text{(Mu)}$$

$$\langle t, 0, 0, \odot, \eta, \Phi_r, \phi\rangle_{\mathsf{tm}} \xrightarrow{\circledR} \langle \Phi_r, (\ulcorner t \urcorner :: \phi + 1) :: \epsilon_{\mathsf{f}}, \eta, \ulcorner t \urcorner :: \phi + 1\rangle_{\mathsf{envfn}} \qquad \text{(Init-Rev)}$$

$$\langle \pi :: \Phi_r, \Phi, \eta, \pi\rangle_{\mathsf{envfn}} \xrightarrow{\circledR} \langle \Phi_r, \pi :: \Phi, \eta, \pi\rangle_{\mathsf{envfn}} \qquad \text{(Rev)}$$

$$\langle \epsilon_{\mathsf{f}}, \Phi, (\lambda.s, e, \Delta, \Phi'), \pi\rangle_{\mathsf{envfn}} \xrightarrow{\bigstar} \langle \lambda.s, e, \Delta, \Phi, \pi\rangle_{\mathsf{ev}} \qquad \text{(Restart)}$$

Figure 13: AB machine for the call-by-name $\lambda\mu$-calculus

restarts the machine, executing $\lambda.s$ in the stack $\ulcorner t \urcorner :: \phi + 1$: the next step is a β-reduction which produces the configuration we want.

7.4.5. *Full abstraction.* As in λ-calculus, the AB machine flags each step of its argument generation process, and for each generated argument t there is a corresponding sequence of flags $\mathsf{Seq}(t)$. We can prove as in Section 5.2 that applicative bisimilarity and the AB machine equivalence coincide. Besides, the AB machine relies on ingredients and operations (counters, max, *etc.*) that we know how to translate in HOcore in a faithful and deterministic way (see Appendix A), and for which we can prove the following full abstraction theorem.

Theorem 7.5. *For all closed closures* (t, e, Δ, Φ) *and* (s, d, Γ, Φ) *of rank* ϕ, $(t, e, \Delta, \Phi) \approx_{\mathsf{app}} (s, d, \Gamma, \Phi)$ *iff* $[\![\langle t, e, \Delta, \Phi, \phi\rangle_{\mathsf{ev}}]\!] \approx_{\mathsf{HO}} [\![\langle s, d, \Gamma, \Phi, \phi\rangle_{\mathsf{ev}}]\!]$.

Full abstraction can also be achieved by internalizing contextual equivalence directly. The resulting machine would extend the machine of Figure 6 with the steps generating μ-binders of Figure 13.

8. Conclusion and Future Work

We propose encodings of the call-by-name and call-by-value λ-calculus with or without control operators into HOcore, fully abstract w.r.t. normal-form and applicative bisimilarities, and contextual equivalence. This shows that a minimal higher-order calculus with a fixed number of hidden names, which is much less expressive than the name-passing π-calculus, still has enough expressive power to faithfully encode these calculi.

We use abstract machines not only to fix the reduction strategy, but also as an intermediary step between the λ-calculus and HOcore. We turn the equivalences of the λ-calculus, and their potentially complex testing conditions, into a first-order bisimilarity over an LTS (a flag-generating machine), which is closer to the HOcore equivalence. We believe this internalization technique can be applied to any language for which an abstract machine has been defined. No matter how intricate the bisimilarities for such a language are, it should be always possible to generate a context as in Section 5.4 to internalize contextual equivalence.

The encodings of the extended abstract machines into HOcore rely on the same principles, e.g., to represent stacks, non-deterministic choice, case analyses on terms, *etc.* We believe it is possible to automatically derive the encoding from an abstract machine so that the generated translation verify Lemmas 4.8 and 4.10, giving us Theorem 4.14 for free.

Our encodings are *weak* compositional [Gor10, Par06], i.e., they consist of a compositional translation of the λ-term inside a fixed process representing the machine. We conjecture there is no fully abstract compositional encoding of the λ-calculus into HOcore. In such an encoding, a translated λ-abstraction could be composed with an argument (a translated λ-term) to β-reduce. It would require at least one communication on a public channel name, but the λ-abstraction should still be able to protect itself from unwanted behaviours from the outside. It seems difficult if not impossible to achieve with only hidden names and no name restriction.

As demonstrated in other settings [Pre22, JS22], our encodings could be useful to import proof techniques such as up-to techniques [SR12] from the process calculus world into the λ-calculus world. It would be interesting especially for applicative bisimilarity for which powerful up-to techniques have not been defined yet.

Finally, we would like to explore further the expressiveness of process calculi by translating the λ-calculus without a predefined reduction strategy. The encodings defined so far in the literature (cf. Section 1) assumes a given strategy for the λ-calculus. An exception is the work by Cai and Fu [CF11], where a λ-term is represented as a tree, and β-reduction is a transformation on trees. It relies on an unbounded number of restricted names to represent tree nodes; we wonder if we can use the same ideas with only a fixed number of names. However, such an encoding works for any source language, as it is a manipulation of syntax trees. We would like the encoding to be more tailored to the λ-calculus, to tell us more about the relationship between the λ-calculus and process calculi.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful comments. This work has been partially funded by PHC Polonium and by the National Science Centre of Poland under grants no. 2014/15/B/ST6/00619 and 2019/33/B/ST6/00289. Damien Pous was supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program

"Investissements d'Avenir" (ANR-11-IDEX- 0007) operated by the French National Research Agency (ANR).

References

[ABDM03] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.

[Acc13] Beniamino Accattoli. Evaluating functions as processes. In Rachid Echahed and Detlef Plump, editors, *Proceedings 7th International Workshop on Computing with Terms and Graphs, TERM-GRAPH 2013*, volume 110 of *EPTCS*, pages 41–55, Rome, Italy, March 2013.

[AH08] Zena M. Ariola and Hugo Herbelin. Control reduction theories: the benefit of structural substitution. *Journal of Functional Programming*, 18(3):373–419, 2008.

[Ama11] Roberto M. Amadio. A decompilation of the pi-calculus and its application to termination. *CoRR*, abs/1102.2339, 2011.

[AO93] Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.

[Bar84] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.

[BBL+17] Małgorzata Biernacka, Dariusz Biernacki, Sergueï Lenglet, Piotr Polesiuk, Damien Pous, and Alan Schmitt. Fully abstract encodings of λ-calculus in hocore through abstract machines. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005118.

[Bef11] Emmanuel Beffara. Functions as proofs as processes. *CoRR*, abs/1107.4160, 2011.

[BHY01] Martin Berger, Kohei Honda, and Nobuko Yoshida. Sequentiality and the pi-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 29–45, Kraków, Poland, May 2001. Springer-Verlag.

[BL14] Dariusz Biernacki and Sergueï Lenglet. Applicative bisimilarities for call-by-name and call-by-value $\lambda\mu$-calculus. In Bart Jacobs, Alexandra Silva, and Sam Staton, editors, *Proceedings of the 30th Annual Conference on Mathematical Foundations of Programming Semantics(MFPS XXX)*, volume 308 of *ENTCS*, pages 49–64, Ithaca, NY, USA, June 2014.

[CCS10] Matteo Cimini, Claudio Sacerdoti Coen, and Davide Sangiorgi. Functions as processes: Termination and the $\overline{\lambda}\mu\tilde{\mu}$-calculus. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *Trustworthly Global Computing - 5th International Symposium, TGC 2010, Revised Selected Papers*, volume 6084 of *Lecture Notes in Computer Science*, pages 73–86, Munich, Germany, February 2010. Springer.

[CF11] Xiaojuan Cai and Yuxi Fu. The λ-calculus in the π-calculus. *Mathematical Structures in Computer Science*, 21(5):943–996, 2011.

[DHS22] Adrien Durier, Daniel Hirschkoff, and Davide Sangiorgi. Eager functions as processes. *Theoretical Computer Science*, 913:8–42, 2022.

[DMAV14] Paul Downen, Luke Maurer, Zena M. Ariola, and Daniele Varacca. Continuations, processes, and sharing. In Olaf Chitil, Andy King, and Olivier Danvy, editors, *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, 2014*, pages 69–80, Kent, United Kingdom, September 2014. ACM.

[FF86] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[FFF09] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[Gor10] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, 2010.

[HYB14] Kohei Honda, Nobuko Yoshida, and Martin Berger. Process types as a descriptive tool for interaction - control and the pi-calculus. In Gilles Dowek, editor, *Rewriting and Typed Lambda*

Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014. Proceedings, volume 8560 of *Lecture Notes in Computer Science*, pages 1–20, Vienna, Austria, July 2014. Springer.

[Jab15] Guilhem Jaber. Operational nominal game semantics. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2015.

[JS22] Guilhem Jaber and Davide Sangiorgi. Games, mobile processes, and functions. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[Kri07] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[Las99a] Søren B. Lassen. Bisimulation for pure untyped λμ-caluclus (extended abstract). Unpublished note, January 1999.

[Las99b] Søren B. Lassen. Bisimulation in untyped lambda calculus: Böhm trees and bisimulation up to context. In Michael Mislove Stephen Brookes, Achim Jung and Andre Scedrov, editors, *Proceedings of the 15th Annual Conference on Mathematical Foundations of Programming Semantics*, volume 20 of *ENTCS*, pages 346–374, New Orleans, LA, April 1999.

[Las05] Søren B. Lassen. Eager normal form bisimulation. In Prakash Panangaden, editor, *Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 345–354, Chicago, IL, June 2005.

[Las06] Søren B. Lassen. Head normal form bisimulation for pairs and the λμ-calculus. In Rajeev Alur, editor, *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS 2006)*, pages 297–306, Seattle, WA, USA, August 2006.

[Lév75] Jean-Jacques Lévy. An algebraic interpretation of the λβK-calculus and a labeled λ-calculus. In Corrado Böhm, editor, *Lambda-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, Italy, March 25-27, 1975*, volume 37 of *Lecture Notes in Computer Science*, pages 147–165. Springer, 1975.

[LPSS10] Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi, and Alan Schmitt. On the expressiveness of polyadic and synchronous communication in higher-order process calculi. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Proceedings, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 442–453, Bordeaux, France, July 2010. Springer.

[LPSS11] Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi, and Alan Schmitt. On the expressiveness and decidability of higher-order process calculi. *Information and Computation*, 209(2):198–226, 2011.

[LS14] Paul Blain Levy and Sam Staton. Transition systems over games. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 64:1–64:10. ACM, 2014.

[Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Ong88] Chih-Hao Luke Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functionnal Programming*. PhD thesis, University of London, 1988.

[Par92] Michel Parigot. λμ-calculus: an algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, number 624 in LNAI, pages 190–201, St. Petersburg, Russia, July 1992.

[Par06] Joachim Parrow. Expressiveness of process algebras. In Catuscia Palamidessi and Frank Valencia, editors, *Proceedings of the LIX Colloquium on Emerging Trends in Concurrency Theory, LIX 2006, Paris, France, November 13-15, 2006*, volume 209 of *Electronic Notes in Theoretical Computer Science*, pages 173–186. Elsevier, 2006.

[Pre22] Enguerrand Prebet. Functions and references in the pi-calculus: Full abstraction and proof techniques. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 130:1–130:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[San92] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. In Andre Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science (LICS'92)*, pages 102–109, Santa Cruz, California, June 1992.

[San96] Davide Sangiorgi. Bisimulation for higher-order process calculi. *Information and Computation*, 131(2):141–178, 1996.

[San99] Davide Sangiorgi. From lambda to pi; or, rediscovering continuations. *Mathematical Structures in Computer Science*, 9(4):367–401, 1999.

[Sau05] Alexis Saurin. Separation with streams in the $\lambda\mu$-calculus. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pages 356–365. IEEE Computer Society, 2005.

[SR98] Thomas Streicher and Bernhard Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.

[SR12] Davide Sangiorgi and Jan J. M. M. Rutten, editors. *Advanced Topics in Bisimulation and Coinduction*, volume 52 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 2012.

[SW01] Davide Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[TCP12] Bernardo Toninho, Luís Caires, and Frank Pfenning. Functions as session-typed processes. In Lars Birkedal, editor, *Foundations of Software Science and Computation Structures, 15th International Conference (FOSSACS'12)*, number 7213 in LNCS, pages 346–360, Tallinn, Estonia, March 2012.

[vBV14] Steffen van Bakel and Maria Grazia Vigliotti. A fully-abstract semantics of lambda-mu in the pi-calculus. In Paulo Oliva, editor, *Proceedings Fifth International Workshop on Classical Logic and Computation, CL&C 2014*, volume 164 of *EPTCS*, pages 33–47, Vienna, Austria, July 2014.

[YBH04] Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the pi-calculus. *Information and Computation*, 191(2):145–202, 2004.

Appendix A. Translation of the $\lambda\mu$-calculus AB Machine into HOcore

The translation of the AB machine for the call-by-name $\lambda\mu$-calculus extends the one for the plain λ-calculus of Section 5.3. We omit the parts of the translation which are the same or similar to the plain λ-calculus one.

The translation of the evaluation mode is as follows. A closure is now composed of four messages. Because the closures are supposed to be closed, the process representing the empty environments ϵ_{t} and ϵ_{b} should never be executed, and we choose to represent them with $\mathbf{0}$. The definitions of $[\![\epsilon_{\mathsf{f}}]\!]$ and $[\![\alpha]\!]$ are discussed later.

$$[\![\langle t, e, \Delta, \Phi, \pi\rangle_{\mathsf{ev}}]\!] \triangleq [\![t]\!] \parallel \overline{et}\langle[\![e]\!]\rangle \parallel \overline{eb}\langle[\![\Delta]\!]\rangle \parallel \overline{ef}\langle[\![\Phi]\!]\rangle \parallel \overline{c}\langle[\![\pi]\!]\rangle \parallel P_{\mathsf{rec}}$$

$$[\![(t, e, \Delta, \Phi)]\!] \triangleq \overline{\eta_1}\langle[\![t]\!]\rangle \parallel \overline{\eta_2}\langle[\![e]\!]\rangle \parallel \overline{\eta_3}\langle[\![\Delta]\!]\rangle \parallel \overline{\eta_4}\langle[\![\Phi]\!]\rangle$$

$$[\![\eta :: e]\!] \triangleq \overline{hd_{\mathsf{e}}}\langle[\![\eta]\!]\rangle \parallel \overline{et}\langle[\![e]\!]\rangle \qquad [\![\epsilon_{\mathsf{t}}]\!] \triangleq \mathbf{0}$$

$$[\![\pi :: \Delta]\!] \triangleq \overline{hd_{\mathsf{b}}}\langle[\![\pi]\!]\rangle \parallel \overline{eb}\langle[\![\Delta]\!]\rangle \qquad [\![\epsilon_{\mathsf{b}}]\!] \triangleq \mathbf{0}$$

$$[\![\pi :: \Phi]\!] \triangleq \overline{hd_{\mathsf{f}}}\langle[\![\pi]\!]\rangle \parallel \overline{ef}\langle[\![\Phi]\!]\rangle \qquad [\![\epsilon_{\mathsf{f}}]\!] \triangleq \dots$$

$$[\![\eta :: \pi]\!] \triangleq \overline{hd_{\mathsf{c}}}\langle[\![\eta]\!]\rangle \parallel \overline{c}\langle[\![\pi]\!]\rangle \qquad [\![\alpha]\!] \triangleq \dots$$

The representations of λ-abstractions, applications and indices are the same as in plain λ-calculus, adapted to the fact that closures are composed of four elements now. A μ operator captures the current stack and extend the bound-names environment Δ accordingly.

$$[\![\mu.[\alpha]t]\!] \triangleq \mathsf{Mu}_{\mathsf{ev}}([\![\alpha]\!], [\![t]\!]) \qquad \mathsf{Mu}_{\mathsf{ev}}(P_\alpha, P_t) \triangleq c(x).eb(y).\left(P_t \parallel \overline{eb}\left\langle\overline{hd_{\mathsf{b}}}\langle x\rangle \parallel \overline{eb}\langle y\rangle\right\rangle \parallel \overline{c}\langle P_\alpha\rangle\right)$$

The translation $[\![\delta]\!]$ implements the mode bn resolving bound names. The translation $[\![\delta]\!]_{\mathsf{b}}$ implements the search of δ in Δ, while $[\![\delta]\!]$ make a save of Δ before starting the search. The environment Δ is restored by $[\![0]\!]_{\mathsf{b}}$ when the search is over.

$$[\![\delta]\!] \triangleq \mathsf{Bound}_{\mathsf{ev}}([\![\delta]\!]_{\mathsf{b}}) \qquad \mathsf{Bound}_{\mathsf{ev}}(P_\delta) \triangleq eb(x).(P_\delta \parallel \overline{eb}\langle x\rangle \parallel \overline{sv_{\mathsf{b}}}\langle x\rangle)$$

$$[\![\delta+1]\!]_{\mathsf{b}} \triangleq \mathsf{IdMu}_{\mathsf{ev}}([\![\delta]\!]_{\mathsf{b}}) \qquad \mathsf{IdMu}_{\mathsf{ev}}(P_\delta) \triangleq eb(x).(x \parallel hd_{\mathsf{b}}(\_).P_\delta)$$

$$[\![0]\!]_{\mathsf{b}} \triangleq eb(x).(x \parallel hd_{\mathsf{b}}(y).eb(\_).sv_{\mathsf{b}}(z).(y \parallel \overline{eb}\langle z\rangle))$$

Similarly, the translation $[\![\phi]\!]$ implements the mode fn resolving free names. The translation $[\![\phi]\!]_{\mathsf{f}}$ implements the search of ϕ in Φ, while $[\![\phi]\!]$ make a save of Φ (on sv_{f}) and ϕ (on sv_ϕ) before starting the search. We compute the reverse of Φ on the name rev using the processes MtRev and $\mathsf{ConsRev}$, which basically encode a stack. If the search succeeds, the messages on rev on sv_ϕ are not useful and are discarded, and the save of Φ is restored.

$$\mathsf{MtRev} \triangleq mtrev(x).csrev(\_).x$$

$$\mathsf{ConsRev}(P_\pi, P_\Phi) \triangleq \overline{hd_{\mathsf{rev}}}\langle P_\pi\rangle \parallel \overline{rev}\langle P_\Phi\rangle \parallel mtrev(\_).csrev(x).x$$

$$[\![\phi]\!] \triangleq \mathsf{Free}([\![\phi]\!]_{\mathsf{f}}) \quad \mathsf{Free}(P_\phi) \triangleq ef(x).(P_\phi \parallel \overline{ef}\langle x\rangle \parallel \overline{sv_{\mathsf{f}}}\langle x\rangle \parallel \overline{sv_\phi}\langle P_\phi\rangle \parallel \overline{rev}\langle\mathsf{MtRev}\rangle)$$

$$[\![\phi+1]\!]_{\mathsf{f}} \triangleq \mathsf{IdF}([\![\phi]\!]_{\mathsf{f}}) \quad \mathsf{IdF}(P_\phi) \triangleq ef(x).rev(y).(x \parallel hd_{\mathsf{f}}(z).(P_\phi \parallel \overline{rev}\langle\mathsf{ConsRev}(z, y)\rangle))$$

$$[\![0]\!]_{\mathsf{f}} \triangleq ef(x).(x \parallel hd_{\mathsf{f}}(y).ef(\_).sv_{\mathsf{f}}(z).sv_\phi(\_).rev(\_).(y \parallel \overline{ef}\langle z\rangle))$$

If the search fails, it means that ϕ is not mapped to a stack, and $[\![\epsilon_{\mathsf{f}}]\!]$ should start generating the argument. It first discards the message on sv_{f} which is no longer useful and sets up the ind mode.

$$[\![\epsilon_{\mathsf{f}}]\!] \triangleq \circledcirc(\_).sv_{\mathsf{f}}(\_).(\overline{ind}\langle[\![0]\!]\rangle \parallel \overline{k}\langle[\![1]\!]_{\mathsf{c}}\rangle \parallel \overline{km}\langle[\![0]\!]_{\mathsf{cm}}\rangle \parallel \overline{r}\langle[\![\odot]\!]\rangle \parallel \overline{initInd}\langle\mathbf{0}\rangle)$$

The message on km is for the counter κ_μ, and the translation $[\![\kappa_\mu]\!]_{\mathsf{cm}}$ is the same as $[\![\kappa]\!]_{\mathsf{c}}$, but using different names $zerom$ and $sucm$. The RecTm process generating terms needs an extra case analysis on κ_μ, as the term generation process may end only if $\kappa_\mu = 0$. The process RecTm of Section 5.3 is modified to do this case analysis when $\kappa = 0$.

$$\mathsf{RecTm} \triangleq \ldots$$

$$\overline{zero}\left\langle km(u).\left(\begin{array}{l} u \parallel \overline{zerom}\left\langle \begin{array}{l}(\mathsf{Lambda}(z) \parallel \overline{r}\langle[\![\odot]\!]\rangle \parallel \overline{km}\langle u\rangle) + (\mathsf{AppFun}(z,[\![\odot]\!]) \parallel \overline{km}\langle u\rangle)\\ + (\mathsf{Mu} \parallel \overline{r}\langle[\![\odot]\!]\rangle \parallel \overline{k}\langle z\rangle \parallel \overline{km}\langle u\rangle) + \mathsf{DoneTm}\end{array}\right\rangle \\ \quad \parallel \overline{sucm}\left\langle sukm(\_).\left(\begin{array}{l}(\mathsf{Lambda}(z) \parallel \overline{r}\langle[\![\odot]\!]\rangle \parallel \overline{km}\langle u\rangle)\\ +(\mathsf{AppFun}(z,[\![\odot]\!]) \parallel \overline{km}\langle u\rangle)\\ +(\mathsf{Mu} \parallel \overline{r}\langle[\![\odot]\!]\rangle \parallel \overline{k}\langle z\rangle \parallel \overline{km}\langle u\rangle)\end{array}\right)\right\rangle \end{array}\right)\right\rangle$$

Generating a de Bruijn index for a term variable, a λ-abstraction, or an application is done the same way as in plain λ-calculus: the processes RecInd, Succ, Var, Lambda, AppFun, and App are the same as in Section 5.3.

The process Mu generates a μ binder, starting with the bound name δ, simulating the indMu mode. It initiates a counter on $km2$, which is increased in parallel with δ. That counter is then compared in MuTm with the one on km to compute the maximum between them, using the same technique as in App but on different names. The processes RecIMu and SuccMu behave like their respective counterpart RecInd and Succ.

$$\mathsf{Mu} \triangleq i\mu(\_).(\overline{indMu}\langle[\![0]\!]_{\mathsf{b}}\rangle \parallel \overline{km2}\langle[\![1]\!]_{\mathsf{cm}}\rangle \parallel \overline{initIMu}\langle\mathbf{0}\rangle)$$

$$\mathsf{RecIMu} \triangleq initIMu(\_).recimu(x).(x \parallel \overline{recimu}\langle x\rangle \parallel \mathsf{SuccMu} + \mathsf{MuTm})$$

$$\mathsf{SuccMu} \triangleq s\mu(\_).indMu(x).km2(y).(\overline{indMu}\langle\mathsf{IdMu}_{\mathsf{ev}}(x)\rangle \parallel \overline{km2}\langle\mathsf{Sukm}(y)\rangle \parallel \overline{initIMu}\langle\mathbf{0}\rangle)$$

$$\mathsf{MuTm} \triangleq \mu(\_).tm(x).km(y_1).km2(y_2).indMu(z).\left(\begin{array}{l}\overline{maxm1}\langle y_1\rangle \parallel \overline{maxm2}\langle y_2\rangle \parallel \overline{initm1}\langle y_1\rangle \parallel \overline{initm2}\langle y_2\rangle \parallel \\ resu(y).\left(\begin{array}{l}\overline{tm}\langle\mathsf{Mu}_{\mathsf{ev}}(\mathsf{Bound}_{\mathsf{ev}}(z),x)\rangle \parallel y \parallel \\ sukm(y').(\overline{zerom}\langle\mathbf{0}\rangle \parallel \overline{sucm}\langle\overline{km}\langle y'\rangle \parallel \overline{initTm}\langle\mathbf{0}\rangle\rangle)\end{array}\right)\end{array}\right)$$

The process DoneTm is executed when we are done computing the term t and start computing the new testing environment Φ. Using the process representing ϕ saved on the name sv_ϕ, it computes the stack at the bottom of this new environment $\ulcorner t \urcorner :: \phi + 1$, saves it on the name sv_π, and put it also on the name $newf$ on which Φ is computed.

$$\mathsf{DoneTm} \triangleq \text{®}(\_).tm(x).sv_\phi(y).\left(\begin{array}{l}\overline{newf}\langle\overline{hd_{\mathsf{f}}}\langle\mathsf{Stk}(x,y)\rangle \parallel \overline{ef}\langle[\![\epsilon_{\mathsf{f}}]\!]\rangle\rangle \parallel \\ \overline{sv_\pi}\langle\mathsf{Stk}(x,y)\rangle \parallel \overline{initRev}\langle\mathbf{0}\rangle\end{array}\right)$$

$$\mathsf{Stk}(P_t,P_\phi) \triangleq \overline{hd_{\mathsf{c}}}\langle\overline{\eta_1}\langle P_t\rangle \parallel \overline{\eta_2}\langle[\![\epsilon_{\mathsf{t}}]\!]\rangle \parallel \overline{\eta_3}\langle[\![\epsilon_{\mathsf{b}}]\!]\rangle \parallel \overline{\eta_4}\langle[\![\epsilon_{\mathsf{f}}]\!]\rangle\rangle \parallel \overline{c}\langle\mathsf{Free}(\mathsf{IdF}(P_\phi))\rangle$$

The process RecRev computes Φ on $newf$ by reversing the environment that hs been saved on rev. It proceeds by case analysis on the process on rev, which is either encoding a stack constructor or an empty stack.

$$\mathsf{RecRev} \triangleq initRev(\_).recrev(x).(x \parallel \overline{recrev}\langle x\rangle \parallel rev(y).newf(z).\left(\begin{array}{l} y \parallel \overline{csrev}\Big\langle \circledR(\_).hd_{\mathsf{rev}}(u).(\overline{newf}\big\langle \overline{hd_{\mathsf{f}}}\langle u\rangle \parallel \overline{ef}\langle z\rangle\big\rangle \parallel \overline{initRev}\langle \mathbf{0}\rangle)\Big\rangle \\ \quad \parallel \overline{mtrev}\langle \mathsf{Done}(z)\rangle \end{array}\right)\Big)$$

During the argument generation, the process $[\![0]\!]_{\mathsf{f}}$ was stuck expecting several messages, which can now be provided by the process Done.

$$\mathsf{Done}(P_\Phi) = \bigstar(\_).sv_\pi(x).(\overline{hd_{\mathsf{f}}}\langle z\rangle \parallel \overline{ef}\langle \mathbf{0}\rangle \parallel \overline{sv_{\mathsf{f}}}\langle P_\Phi\rangle \parallel \overline{sv_\phi}\langle \mathbf{0}\rangle \parallel \overline{rev}\langle \mathbf{0}\rangle)$$