Fully Abstract Encodings of λ-Calculus in HOcore through Abstract Machines

Małgorzata Biernacka*, Dariusz Biernacki*, Sergueï Lenglet†, Piotr Polesiuk*, Damien Pous‡ and Alan Schmitt ○
* University of Wrocław † Université de Lorraine ‡ Univ. Lyon, CNRS, ENS de Lyon, UCB Lyon 1 ○ Inria

***Abstract*—We present fully abstract encodings of the call-by-name 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.**

I. Introduction

HOcore is a minimal process calculus with higher-order communication, meaning that messages are executable processes. It is a subcalculus of HOπ [19] with no construct to generate names or to restrict the scope of communication channels. Even with such a limited syntax, HOcore is Turing complete [16]. 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 [15], while it can be done in π [21].

Although HOcore is Turing complete, we initially thought a fully abstract encoding of the λ-calculus into HOcore was impossible. 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 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 a novel encoding of the call-by-name λ-calculus, or more precisely the Krivine Abstract Machine (KAM) [14], 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* [16], 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* [17], where normal forms are decomposed into subterms that must be bisimilar. Next, we turn to *applicative bisimilarity* [1], 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 KAM, which performs the evaluation of a term, with additional transitions with flags interpreting the equivalences. By doing so, we can express these different equivalences on terms by a simpler bisimilarity on these flags-generating machines, which can be seen as labeled transition systems (LTS). Finally, we translate these extended machines into HOcore and prove full abstraction for all three equivalences. Altogether, this work shows that a minimal process calculus with no name restriction can faithfully encode the call-by-name λ-calculus.

The chosen equivalences: We study normal-form and applicative bisimilarities, even though faithfully encoding contextual equivalence is enough to get full abstraction. Our motivation is twofold. 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 then 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 [18], other encodings of λ into π have been proposed either as a result itself [9], or to study other properties such as connections with logic [2, 5, 22], termination [10, 4, 24], sequentiality [6], control [10, 13, 23], or Continuation-Passing Style (CPS) transforms [20, 21, 11]. These works use the more expressive *first-order* π-calculus, except for [20, 21], discussed below; full abstraction is proved w.r.t. contextual equivalence in [6, 24, 13], normal-form bisimilarity in [23], applicative bisimilarity in [9], and both bisimilarities in [21]. The encodings of [6, 24, 13] are driven by types, and therefore cannot be compared to our untyped setting. In [23], 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 IV-B. The encoding of [9] 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 [20, 21], but it is an abstraction-passing calculus, which is strictly more expressive than a

978-1-5090-3018-7/17/$31.00 ©2017 European Union

process-passing calculus [15]. 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 [3], 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 [21, Chapter 18] but not w.r.t. applicative bisimilarity [21, 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 a translated λ-term. The encoding of [21] does not protect itself against this unwanted behavior from the environment. In contrast, the encoding of Section V and the one in [9] are fully abstract w.r.t. applicative bisimilarity, because they take this issue into account, as we explain in Section V-B.

Outline: Section II presents HOcore. Section III shows how to encode the KAM; the machine and its encoding are then modified to get full abstraction with relation to normal-form bisimilarity (Section IV) and applicative bisimilarity (Section V). Section VI concludes this paper, and an accompanying research report [8] contains the main proofs.

II. The Calculus HOcore

Syntax and semantics: HOcore [16] is a simpler version of HOπ [19] 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:

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

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 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 write $\mathsf{fn}(P)$ for the 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 use label a for an input where the received process does not matter (e.g., $a(\_).P$).

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

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

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

Fig. 1: HOcore LTS

Barbed equivalence: We let γ range over names a and *conames* $\overline{a}$; we define observable actions as follows.

Definition 1. *The process P has a strong observable action on a (resp. $\overline{a}$), written $P \downarrow_a$ (resp. $P \downarrow_{\overline{a}}$), if $P \xrightarrow{a(Q)} R$ (resp. $P \xrightarrow{\overline{a}\langle Q\rangle} R$) for some Q, R. A process P has a weak observable action on γ, written $P \Downarrow_\gamma$, if $P \overset{\tau}{\Longrightarrow} P' \downarrow_\gamma$ for some P'. We write $\mathsf{WkObs}(P)$ for the set of weak observable actions of P.*

Our definition of barbed equivalence depends on a 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 [16], 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.

Definition 2. *A symmetric relation $\mathcal{R}$ is a barbed bisimulation w.r.t. $\mathbb{H}$ if $P \mathrel{\mathcal{R}} Q$ implies*

- *$P \downarrow_\gamma$ and $\gamma \notin \mathbb{H}$ implies $Q \Downarrow_\gamma$;*
- *for all R such that $\mathsf{fn}(R) \cap \mathbb{H} = \varnothing$, we have $P \| R \mathrel{\mathcal{R}} Q \| R$;*
- *if $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \overset{\tau}{\Longrightarrow} Q'$ and $P' \mathrel{\mathcal{R}} Q'$.*

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 $\Downarrow_\gamma$ with $\downarrow_\gamma$ in the first item, and $\overset{\tau}{\Longrightarrow}$ with $\xrightarrow{\tau}$ in the third. From [16], we know that strong barbed equivalence is decidable when $\mathbb{H} = \varnothing$, but undecidable when $\mathbb{H}$ is of cardinal at least 4. We lower this bound to 2 in Theorem 4.

III. 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.

A. Definition of the KAM

The KAM [14] 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.

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

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

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 $[s/x]t$. If a configuration of the form $\lambda x.t \star []$ 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 $x \star \pi$.

B. 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 n is always encoded as a message on hd_n for its head and one on n 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 $\overline{hd_\mathsf{c}}\langle 1 \rangle \parallel \overline{c}\Big\langle \overline{hd_\mathsf{c}}\langle 2 \rangle \parallel \overline{c}\Big\langle \overline{hd_\mathsf{c}}\langle 3 \rangle \parallel \overline{c}\Big\langle \overline{hd_\mathsf{c}}\langle 4 \rangle \parallel \overline{c}\langle \overline{b}\langle \mathbf{0} \rangle \rangle \Big\rangle \Big\rangle \Big\rangle$.

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 for the stacks, terms, and configurations as follows, where we assume the variable p does not occur in the translated entities.

$$\begin{aligned}
[\![t \star \pi]\!] &\triangleq [\![t]\!] \parallel \overline{c}\langle [\![\pi]\!] \rangle \\
[\![[]]\!] &\triangleq \overline{b}\langle \mathbf{0} \rangle \\
[\![t :: \pi]\!] &\triangleq \overline{hd_\mathsf{c}}\langle [\![t]\!] \rangle \parallel \overline{c}\langle [\![\pi]\!] \rangle \\
[\![t\, s]\!] &\triangleq c(p).\big([\![t]\!] \parallel \overline{c}\big\langle \overline{hd_\mathsf{c}}\langle [\![s]\!] \rangle \parallel \overline{c}\langle p \rangle \big\rangle\big) \\
[\![\lambda x.t]\!] &\triangleq c(p).(hd_\mathsf{c}(x).\, [\![t]\!] \parallel p) \\
[\![x]\!] &\triangleq x
\end{aligned}$$

In the translation of a configuration $t \star \pi$, 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. 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 $\overline{hd_\mathsf{c}}\langle [\![s]\!] \rangle \parallel \overline{c}\langle p \rangle$. 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.

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

Theorem 3. *In the forward direction, if $C \mapsto^* C'$, then $[\![C]\!] \xRightarrow{\tau} [\![C']\!]$. In the backward direction, if $[\![C]\!] \xRightarrow{\tau} P$, then there exists a C' such that $C \mapsto^* C'$ and either*

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

Sketch. The proof is straightforward in the forward direction. In the backward direction, we show that the translation is deterministic (if $[\![C]\!] \xRightarrow{\tau} P \xrightarrow{\tau} Q_1$ and $[\![C]\!] \xRightarrow{\tau} P \xrightarrow{\tau} Q_2$, 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. □

We can then improve over the result of [16] about undecidability of strong barbed equivalence by hiding hd_c and c.

Theorem 4. *Strong barbed equivalence is undecidable in HOcore with 2 hidden names.*

Proof. Assume we can decide strong barbed congruence with two hidden names, and let t be a closed λ-term. We can thus decide if $[\![t \star []]\!]$ is strong barbed-congruent to $c(x).(\, x \parallel \overline{c}\langle x \rangle\,) \parallel \overline{c}\big\langle c(x).(x \parallel \overline{c}\langle x \rangle)\big\rangle$ when hd_c and c are hidden. As the second term loops with no barbs, deciding congruence is equivalent to deciding whether the reduction of t converges, hence a contradiction. □

IV. Normal-Form Bisimilarity

Our first full abstraction result is for normal-form bisimilarity [17]. 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.

A. 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 α, from bound variables, ranged over by x. The grammars of terms (t, s) and values (v) become as follows.

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

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 α is. We write $\mathsf{fv}(t)$ for the set of free variables of t. A variable α is said *fresh* if it does not occur in any entities under consideration.

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

Definition 5. *A symmetric relation $\mathcal{R}$ is a normal-form bisimulation if $t \mathrel{\mathcal{R}} s$ implies:*

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

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

B. Abstract Machine

We now extend the KAM so that it provides additional steps, identified by labeled transitions, that capture the testing done by normal-form bisimilarity. This extended machine features *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 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 $\alpha \star \pi$, 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 [23], 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 [23], 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.

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 α, i.e., a number, then we flag the value of α before entering the next mode (transition VAR).

$$
\begin{array}{rlr}
\langle t\; s, \pi, n\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto} \langle t, s :: \pi, n\rangle_{\mathsf{ev}} & & (\text{PUSH})\\
\langle \lambda x.t, s :: \pi, n\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto} \langle [s \,/\, x]t, \pi, n\rangle_{\mathsf{ev}} & & (\text{GRAB})\\
\langle \lambda x.t, [\,], n\rangle_{\mathsf{ev}} \overset{\circledcirc}{\longmapsto} \langle [n \,/\, x]t, [\,], n+1\rangle_{\mathsf{ev}} & & (\text{LAMBDA})\\
\langle \alpha, \pi, n\rangle_{\mathsf{ev}} \overset{\alpha}{\longmapsto} \langle \pi, n\rangle_{\mathsf{ct}} & & (\text{VAR})\\
\\
\langle [\,], n\rangle_{\mathsf{ct}} \overset{☆}{\longmapsto} & & (\text{DONE})\\
\langle t :: \pi, n\rangle_{\mathsf{ct}} \overset{\blacktriangledown}{\longmapsto} \langle t, [\,], n\rangle_{\mathsf{ev}} & & (\text{ENTER})\\
\langle t :: \pi, n\rangle_{\mathsf{ct}} \overset{\blacktriangleright}{\longmapsto} \langle \pi, n\rangle_{\mathsf{ct}} & & (\text{SKIP})
\end{array}
$$

Fig. 2: NFB Machine

In *continuation mode* $\langle \pi, n\rangle_{\mathsf{ct}}$, the transition DONE simply finishes the execution if $\pi = [\,]$, using the flag ☆. 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 ☆), 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 6. *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.*

$$
\begin{array}{lr}
\quad \langle (\lambda x.x)\ (\lambda y.y\ 0\ \Omega), [\,], 1\rangle_{\mathsf{ev}} & \\
\overset{\tau}{\longmapsto} \langle \lambda x.x, \lambda y.y\ 0\ \Omega :: [\,], 1\rangle_{\mathsf{ev}} & (\text{PUSH})\\
\overset{\tau}{\longmapsto} \langle \lambda y.y\ 0\ \Omega, [\,], 1\rangle_{\mathsf{ev}} & (\text{GRAB})\\
\overset{\circledcirc}{\longmapsto} \langle 1\ 0\ \Omega, [\,], 2\rangle_{\mathsf{ev}} & (\text{LAMBDA})\\
\overset{\tau}{\longmapsto} \langle 1\ 0, \Omega :: [\,], 2\rangle_{\mathsf{ev}} \overset{\tau}{\longmapsto} \langle 1, 0 :: \Omega :: [\,], 2\rangle_{\mathsf{ev}} & (\text{PUSH - PUSH})\\
\overset{1}{\longmapsto} \langle 0 :: \Omega :: [\,], 2\rangle_{\mathsf{ct}} & (\text{VAR})
\end{array}
$$

We then have three possibilities. First, we reduce the top of the stack, with the sequence $\langle 0 :: \Omega :: [\,], 2\rangle_{\mathsf{ct}} \overset{\blacktriangledown}{\longmapsto} \langle 0, [\,], 2\rangle_{\mathsf{ev}} \overset{0}{\longmapsto} \langle [\,], 2\rangle_{\mathsf{ct}} \overset{☆}{\longmapsto}$. Second, we evaluate Ω with the sequence $\langle 0 :: \Omega :: [\,], 2\rangle_{\mathsf{ct}} \overset{\blacktriangleright}{\longmapsto} \langle \Omega :: [\,], 2\rangle_{\mathsf{ct}} \overset{\blacktriangledown}{\longmapsto} \langle \Omega, [\,], 2\rangle_{\mathsf{ev}}$, and then the machine loops without generating any flag. Third, we skip both terms with $\langle 0 :: \Omega :: [\,], 2\rangle_{\mathsf{ct}} \overset{\blacktriangleright}{\longmapsto} \langle \Omega :: [\,], 2\rangle_{\mathsf{ct}} \overset{\blacktriangleright}{\longmapsto} \langle [\,], 2\rangle_{\mathsf{ct}} \overset{☆}{\longmapsto}$. 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 7. *For all t, t', π, π', n, $t \star \pi \mapsto t' \star \pi'$ 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 8. *A symmetric relation $\mathcal{R}$ is a machine bisimulation if $C_1 \mathrel{\mathcal{R}} C_2$ 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 $C_1' \mathrel{\mathcal{R}} C_2'$;*
- *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 9. *We have $t \approx_{\mathsf{nf}} s$ iff there exists $n > \max(\mathsf{fv}(t) \cup \mathsf{fv}(s))$ such that $\langle t, [], n\rangle_{\mathsf{ev}} \approx_{\mathsf{m}} \langle s, [], n\rangle_{\mathsf{ev}}$.*

Sketch. Machine bisimilarity implies $\approx_{\mathsf{nf}}$ because

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

is a normal-form bisimulation, and the other direction is by showing that

$$\{(\langle t, [], n\rangle_{\mathsf{ev}}, \langle s, [], n\rangle_{\mathsf{ev}}) \mid t \approx_{\mathsf{nf}} s, n > \max(\mathsf{fv}(t), \mathsf{fv}(s))\}$$
$$\cup \{(\langle \pi, n\rangle_{\mathsf{ct}}, \langle \pi', n\rangle_{\mathsf{ct}}) \mid \pi \approx_{\mathsf{nf}} \pi', n > \max(\mathsf{fv}(\pi), \mathsf{fv}(\pi'))\}$$

is a machine bisimulation. □

C. 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 $\llbracket . \rrbracket_{\mathsf{Int}}$ 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. Free variables α are also numbers, since we cannot generate new names in HOcore, 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 V-C, which can be encoded in the same way.

A stack is represented as in the KAM, by messages on hd_{c} and c, and the translation of an application $\llbracket t\ s \rrbracket$ is exactly the same as for the KAM. The encoding of the empty context $[]$

$$\begin{aligned}
\llbracket t\ s \rrbracket &\triangleq c(p).\big(\llbracket t \rrbracket \parallel \overline{c}\big\langle \overline{hd_{\mathsf{c}}}\langle \llbracket s \rrbracket\rangle \parallel \overline{c}\langle p\rangle\big\rangle\big)\\
\llbracket \lambda x.t \rrbracket &\triangleq c(p).(p \parallel \overline{b}\langle \mathsf{Restart}\rangle \parallel hd_{\mathsf{c}}(x).b(\_).\llbracket t \rrbracket)\\
\llbracket x \rrbracket &\triangleq x\\
\llbracket \alpha \rrbracket &\triangleq \llbracket \alpha \rrbracket_{\mathsf{Int}}\\[1ex]
\mathsf{Restart} &\triangleq \circledcirc(\_).k(x).\begin{pmatrix} \overline{hd_{\mathsf{c}}}\langle x\rangle \parallel \overline{k}\langle suc(\_).x\rangle \parallel \\ \overline{c}\langle \llbracket [] \rrbracket\rangle \parallel \overline{b}\langle \mathbf{0}\rangle \end{pmatrix}\\
\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 \star(\_).\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 \llbracket [] \rrbracket\rangle) + \blacktriangleright(\_).\overline{init}\langle \mathbf{0}\rangle\\[1ex]
\llbracket [] \rrbracket &\triangleq b(x).x\\
\llbracket t :: \pi \rrbracket &\triangleq \overline{hd_{\mathsf{c}}}\langle \llbracket t \rrbracket\rangle \parallel \overline{c}\langle \llbracket \pi \rrbracket\rangle\\
\llbracket 0 \rrbracket_{\mathsf{Int}} &\triangleq z(\_).\overline{init}\langle \mathbf{0}\rangle\\
\llbracket n+1 \rrbracket_{\mathsf{Int}} &\triangleq suc(\_).\llbracket n \rrbracket_{\mathsf{Int}}\\[1ex]
\llbracket \langle t, \pi, n\rangle_{\mathsf{ev}} \rrbracket &\triangleq \llbracket t \rrbracket \parallel \overline{c}\langle \llbracket \pi \rrbracket\rangle \parallel \overline{k}\langle \llbracket n \rrbracket_{\mathsf{Int}}\rangle \parallel \mathsf{Rec} \parallel \overline{rec}\langle \mathsf{Rec}\rangle\\
\llbracket \langle \pi, n\rangle_{\mathsf{ct}} \rrbracket &\triangleq \overline{c}\langle \llbracket \pi \rrbracket\rangle \parallel \overline{k}\langle \llbracket n \rrbracket_{\mathsf{Int}}\rangle \parallel \mathsf{Cont} \parallel \mathsf{Rec} \parallel \overline{rec}\langle \mathsf{Rec}\rangle
\end{aligned}$$

Fig. 3: Translation of the NFB machine into HOcore

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 $\mathsf{Restart}$. More precisely, $\llbracket \lambda x.t \rrbracket$ receives the current stack $\llbracket \pi \rrbracket$ on c to run it in parallel with $\overline{b}\langle \mathsf{Restart}\rangle \parallel hd_{\mathsf{c}}(x).b(\_).\llbracket t \rrbracket$. If $\llbracket \pi \rrbracket$ is of the form $\overline{hd_{\mathsf{c}}}\langle \llbracket t' \rrbracket\rangle \parallel \overline{c}\langle \llbracket \pi' \rrbracket\rangle$, then we have the same behavior as with the KAM, with an extra communication on b to garbage collect the $\mathsf{Restart}$ process. Otherwise, $\llbracket \pi \rrbracket = b(x).x$ and we obtain the following sequence of transitions.

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

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 α is translated using $[\![.]\!]_{\mathsf{Int}}$, which signals first the value of α (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:

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

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.

D. Operational Correspondence and Full Abstraction

Establishing full abstraction requires first to state the correspondence between the NFB machine and its translation. In what follows, we let f range over the flags ⊚, ▼, ▶, ☆, and α, where α is used as a shorthand for the succession of α suc flags followed by a z flag. We let $\hat{f}$ range over flags and τ.

Definition 10. *A process P is a machine process if there exists a configuration C of the machine such that $[\![C]\!] \xRightarrow{\tau} P$.*

To establish the correspondence between the machine and its translation, we first define a predicate $\mathsf{next}(\hat{f}, P)$ such that, if $Q \in \mathsf{next}(\hat{f}, P)$, then there is a weak reduction with a step $\hat{f}$ from P to Q, Q is the translation of a machine, and there is no machine translation in between. Intuitively, it is the first translation reached after a step $\hat{f}$.

Definition 11. *We write $Q \in \mathsf{next}(\hat{f}, P)$ if $P \xRightarrow{\tau}\xrightarrow{\hat{f}}\xRightarrow{\tau} Q$, $Q = [\![C']\!]$ for some C', and for any P' such that $P' \neq P$, $P' \neq Q$, and $P \xRightarrow{\hat{f}} P' \xRightarrow{\tau} Q$ or $P \xRightarrow{\tau} P' \xRightarrow{\hat{f}} Q$, we have $P' \neq [\![C]\!]$ for any C.*

A machine process is either a translation of a configuration, or an intermediary state between two translations; $\mathsf{next}(\hat{f}, P)$ gives the set of next processes which are translations. We now prove that, with a single exception, the translation is deterministic. The exception to determinism corresponds to the choice made in continuation mode, which is also not deterministic (but flagged) in the machine. We call *choice process* a process about to make that choice; such processes are of the form $\mathsf{Chce}(P_t, P_\pi, P_n) \triangleq \mathsf{Chce}(P_t) \parallel \overline{c}\langle P_\pi\rangle \parallel \overline{k}\langle P_n\rangle \parallel \mathsf{Rec} \parallel \overline{rec}\langle\mathsf{Rec}\rangle$.

Lemma 12. *Let P be a machine process which is not a choice process. If $P \xrightarrow{\hat{f}} P'$ and $P \xrightarrow{\hat{f}'} P''$, then $\hat{f} = \hat{f}'$ and $P' = P''$. Let P be a choice process. If $P \xrightarrow{\tau} P'$, $P \xrightarrow{\tau} P''$, and $\mathsf{WkObs}(P') = \mathsf{WkObs}(P'')$, then $P' = P''$.*

The choice process $\mathsf{Chce}(P_t, P_\pi, P_n)$ may only reduce to the translation of a configuration after a flag ▶ or ▼. Thus $\mathsf{next}(\tau, \mathsf{Chce}(P_t, P_\pi, P_n))$ is empty, and Lemma 12 implies that $\mathsf{next}(\hat{f}, P)$ is a singleton if it is not empty. 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.

We can now state the correspondence between the NFB machine and its translation.

Lemma 13. *The following assertions hold:*

- $C \overset{\hat{f}}{\mapsto} C'$ *iff* $[\![C]\!] \xRightarrow{\hat{f}} [\![C']\!]$ *and* $\mathsf{next}(\hat{f}, [\![C]\!]) = [\![C']\!]$.
- $C \overset{☆}{\mapsto}$ *iff* $[\![C]\!] \xRightarrow{\tau}\xrightarrow{☆} P$ *and* $P \approx_{\mathsf{HO}} \mathbf{0}$.

With this result, we can relate the bisimilarities of each calculus. Henceforth, we write $\approx_{\mathsf{HO}}$ for $\approx_{\mathsf{HO}}^{\mathbb{H}}$ where $\mathbb{H}$ contains the names of the translation that are not flags. We define the process complementing a flag as follows: for $f \neq \alpha$, we write $\overline{f}$ for the process $\overline{f}\langle\mathbf{0}\rangle$, and we write $\overline{\alpha}$ for the process defined as $\overline{n+1} \triangleq \overline{suc}\langle\mathbf{0}\rangle \parallel \overline{n}$ and $\overline{0} \triangleq \overline{z}\langle\mathbf{0}\rangle$.

Theorem 14. $C \approx_{\mathsf{m}} C'$ *iff* $[\![C]\!] \approx_{\mathsf{HO}} [\![C']\!]$.

Sketch. To prove that barbed equivalence implies machine equivalence, we show that $\mathcal{R} \triangleq \{(C, C') \mid [\![C]\!] \approx_{\mathsf{HO}} [\![C']\!]\}$ is a machine bisimulation. Let $C_1 \mathrel{\mathcal{R}} C_1'$, so that $C_1 \overset{\tau}{\mapsto}{}^{*} C_2 \overset{f}{\mapsto} C_3$; then $[\![C_1]\!] \parallel \overline{f} \xRightarrow{\tau} [\![C_2]\!] \parallel \overline{f} \xRightarrow{\tau} [\![C_3]\!]$ by Lemma 13, which in turn implies that there exists P such that $[\![C_1']\!] \parallel \overline{f} \xRightarrow{\tau} P$ and $[\![C_3]\!] \approx_{\mathsf{HO}} P$. In particular, we have $\mathsf{WkObs}([\![C_3]\!]) = \mathsf{WkObs}(P)$, meaning that $\neg(P \downarrow_{\overline{f}})$. Consequently, there exists P' such that $[\![C_1']\!] \xRightarrow{\tau} P'$ and $P' \downarrow_f$. We can prove that $P \approx_{\mathsf{HO}} \mathsf{next}(f, P')$, but by definition of next, there exists C_3' so that $\mathsf{next}(f, P') = [\![C_3']\!]$. As a result, we have $[\![C_3]\!] \approx_{\mathsf{HO}} P \approx_{\mathsf{HO}} [\![C_3']\!]$, i.e., $C_3 \mathrel{\mathcal{R}} C_3'$, and $[\![C_1']\!] \xRightarrow{f} [\![C_3']\!]$, which implies $C_1' \overset{\tau}{\mapsto}{}^{*} \overset{f}{\mapsto} C_3'$, as wished. The case $C_1 \overset{\tau}{\mapsto}{}^{*} C_2 \overset{☆}{\mapsto}$ is similar.

For the reverse implication, we prove that

$$\mathcal{R} \triangleq \left\{ \begin{array}{c} (P_C \parallel R, P_{C'} \parallel R) \;\middle|\; C \approx_{\mathsf{m}} C', \\ \mathsf{WkObs}(P_C) = \mathsf{WkObs}(P_{C'}) \\ P_C \xRightarrow{\tau} [\![C]\!], P_{C'} \xRightarrow{\tau} [\![C']\!] \text{ or} \\ [\![C]\!] \xRightarrow{\tau} P_C, [\![C']\!] \xRightarrow{\tau} P_{C'}, \end{array} \right\}$$
$$\cup \{(P \parallel R, P' \parallel R) \mid P \approx_{\mathsf{HO}} \mathbf{0} \approx_{\mathsf{HO}} P'\}$$

is a barbed bisimulation. The difficult part is to check that $P_C \parallel R \xrightarrow{\tau} P'$ with a communication on a flag f is matched by $P_{C'} \parallel R$. We have $P_C \downarrow_f$, which implies that P_C cannot reduce: a machine process is either reducing or has an observable action. We are therefore in the case $[\![C]\!] \xRightarrow{\tau} P_C$, and $R \downarrow_{\overline{f}}$. Suppose $f \neq ☆$; then $P_C \xrightarrow{f} P_{C_2} \xRightarrow{\tau} [\![C_2]\!]$ for some P_{C_2} and with $[\![C_2]\!] = \mathsf{next}(f, P_C)$. Because a machine process is deterministic (Lemma 12), then in fact $P' = P_{C_2} \parallel R'$ for some R'. We have $[\![C]\!] \xRightarrow{f} [\![C_2]\!]$, so $C \overset{\tau}{\mapsto}{}^{*} \overset{f}{\mapsto} C_2$ holds by Lemma 13. Because $C \approx_{\mathsf{m}} C'$, there exists C'_2 such that $C' \overset{\tau}{\mapsto}^* \overset{f}{\mapsto} C'_2$ and $C_2 \approx_{\mathsf{m}} C'_2$. By Lemma 13, this implies $[\![C']\!] \xRightarrow{f} [\![C'_2]\!]$; in particular, there exists $P_{C'_2}$ such that $[\![C']\!] \xRightarrow{\tau}\xrightarrow{f} P_{C'_2} \xRightarrow{\tau} [\![C'_2]\!]$. By Lemma 12, we have $P_{C'} \xRightarrow{\tau}\xrightarrow{f} P_{C'_2}$, therefore we have $P_{C'} \parallel R \xRightarrow{\tau} P_{C'_2} \parallel R'$, with $P_{C_2} \parallel R' \mathcal{R} P_{C'_2} \parallel R'$, as wished. In the case $f = \star$, we show that we obtain processes in the second set of $\mathcal{R}$. □

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

Corollary 15. *We have $t \approx_{\mathsf{nf}} s$ 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 9 and 14. □

V. 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:

$$\lambda x.t \star [] \overset{s}{\mapsto} [s / x]t \star []$$

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.

A. 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.

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

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 $\mathsf{fv}(t) = \emptyset$, and a closure (t, e) is closed if the number of elements of e is bigger than the highest free variable of t.

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.

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

$$\langle t\, s, e, \pi\rangle_{\mathsf{ev}} \xmapsto{\tau} \langle t, e, (s, e) :: \pi\rangle_{\mathsf{ev}} \qquad (\textsc{Push})$$

$$\langle 0, (t, e) :: d, \pi\rangle_{\mathsf{ev}} \xmapsto{\tau} \langle t, e, \pi\rangle_{\mathsf{ev}} \qquad (\textsc{Zero})$$

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

$$\langle \lambda.t, e, \eta :: \pi\rangle_{\mathsf{ev}} \xmapsto{\tau} \langle t, \eta :: e, \pi\rangle_{\mathsf{ev}} \qquad (\textsc{Grab})$$

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 $\langle n + 1, \epsilon, \pi\rangle_{\mathsf{ev}}$. 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 $\square$ at a variable position; plugging a term t in $\mathcal{C}$ is written $\mathcal{C}[t]$. A context is closed if $\mathsf{fv}(\mathcal{C}) = \emptyset$. Contextual equivalence is then defined as follows.

Definition 16. *Two closed terms t and s are contextually equivalent, written $t \approx_{\mathsf{ctx}} s$, if for all closed contexts $\mathcal{C}$, $\langle \mathcal{C}[t], \epsilon, []\rangle_{\mathsf{ev}} \xmapsto{\tau}^* \langle \lambda.t', e', []\rangle_{\mathsf{ev}}$ for some t' and e' iff $\langle \mathcal{C}[s], \epsilon, []\rangle_{\mathsf{ev}} \xmapsto{\tau}^* \langle \lambda.s', d', []\rangle_{\mathsf{ev}}$ for some s' and d'.*

Contextual equivalence is characterized by applicative bisimilarity [1], which reduces closed terms to values that are then applied to an arbitrary argument.

Definition 17. *A symmetric relation $\mathcal{R}$ on closed closures is an applicative bisimulation if $(t, e) \mathcal{R} (s, d)$ and $\langle t, e, []\rangle_{\mathsf{ev}} \xmapsto{\tau}^* \langle \lambda.t', e', []\rangle_{\mathsf{ev}}$ implies that there exist s' and d' such that $\langle s, d, []\rangle_{\mathsf{ev}} \xmapsto{\tau}^* \langle \lambda.s', d', []\rangle_{\mathsf{ev}}$, 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, and we show in Section V-D the machine for contextual equivalence.

B. 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 [9] 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 de Bruijn index, 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 index 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

$$\rho ::= (t,\kappa) :: \rho \;\big|\; \odot$$

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 to restart the machine in evaluation mode when the argument is finished. We distinguish two kinds of states: the index state $\langle n,\kappa,\rho,\eta\rangle_{\mathsf{ind}}$, 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.

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, which requires at least 1 λ-abstraction above it, and with the empty stack $\odot$. The value of the index can then be increased with the transition SUC, which accordingly also increases the value of κ. When we reach the needed value for the index, the transition VAR switches to the term mode; 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.

$$\begin{array}{rcll}
\langle \lambda.t, e, []\rangle_{\mathsf{ev}} & \xmapsto{\circledcirc} & \langle 0, 1, \odot, (t,e)\rangle_{\mathsf{ind}} & (\text{ARG}) \\[1em]
\langle n,\kappa,\rho,\eta\rangle_{\mathsf{ind}} & \xmapsto{\boxplus} & \langle n+1,\kappa+1,\rho,\eta\rangle_{\mathsf{ind}} & (\text{SUC}) \\
\langle n,\kappa,\rho,\eta\rangle_{\mathsf{ind}} & \xmapsto{\boxdot} & \langle n,\kappa,\rho,\eta\rangle_{\mathsf{tm}} & (\text{VAR}) \\
\langle t,\kappa+1,\rho,\eta\rangle_{\mathsf{tm}} & \xmapsto{\lambda} & \langle \lambda.t,\kappa,\rho,\eta\rangle_{\mathsf{tm}} & (\text{LAMBDA}) \\
\langle t,0,\rho,\eta\rangle_{\mathsf{tm}} & \xmapsto{\lambda} & \langle \lambda.t,0,\rho,\eta\rangle_{\mathsf{tm}} & (\text{LAMBDA0}) \\
\langle t,\kappa,\rho,\eta\rangle_{\mathsf{tm}} & \xmapsto{\dashv\!\shortparallel} & \langle 0,1,(t,\kappa)::\rho,\eta\rangle_{\mathsf{ind}} & (\text{APPFUN}) \\
\langle s,\kappa_1,(t,\kappa_2)::\rho,\eta\rangle_{\mathsf{tm}} & \xmapsto{@} & \langle t\ s,\mathsf{max}(\kappa_1,\kappa_2),\rho,\eta\rangle_{\mathsf{tm}} & (\text{APP}) \\
\langle t,0,\odot,(s,e)\rangle_{\mathsf{tm}} & \xmapsto{\star} & \langle s,(t,\epsilon)::e,[]\rangle_{\mathsf{ev}} & (\text{RESTART})
\end{array}$$

Fig. 4: AB machine: argument generation

Example 18. *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 in the process. 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 $(f_1,\ldots,f_n,f'_1,\ldots,f'_m,f_{n+1},\ldots,f_l)$.

Definition 19. *Given a term t, we define $\mathsf{Seq}(t)$ as*

$$\begin{aligned}
\mathsf{Seq}(t) &\triangleq (\mathsf{SeqTm}(t),\star) \\
\mathsf{SeqTm}(t\ s) &\triangleq (\mathsf{SeqTm}(t),\dashv\!\shortparallel,\mathsf{SeqTm}(s),@) \\
\mathsf{SeqTm}(\lambda.t) &\triangleq (\mathsf{Seq}(t),\lambda) \\
\mathsf{SeqTm}(n) &\triangleq (\mathsf{SeqInd}(n),\boxdot) \\
\mathsf{SeqInd}(0) &\triangleq () \\
\mathsf{SeqInd}(n+1) &\triangleq (\mathsf{SeqInd}(n),\boxplus)
\end{aligned}$$

We write $C \xmapsto{\mathsf{Seq}(t)} C'$ for $C \xmapsto{f_1} \ldots \xmapsto{f_m} C'$ where $\mathsf{Seq}(t) = (f_1,\ldots,f_m)$.

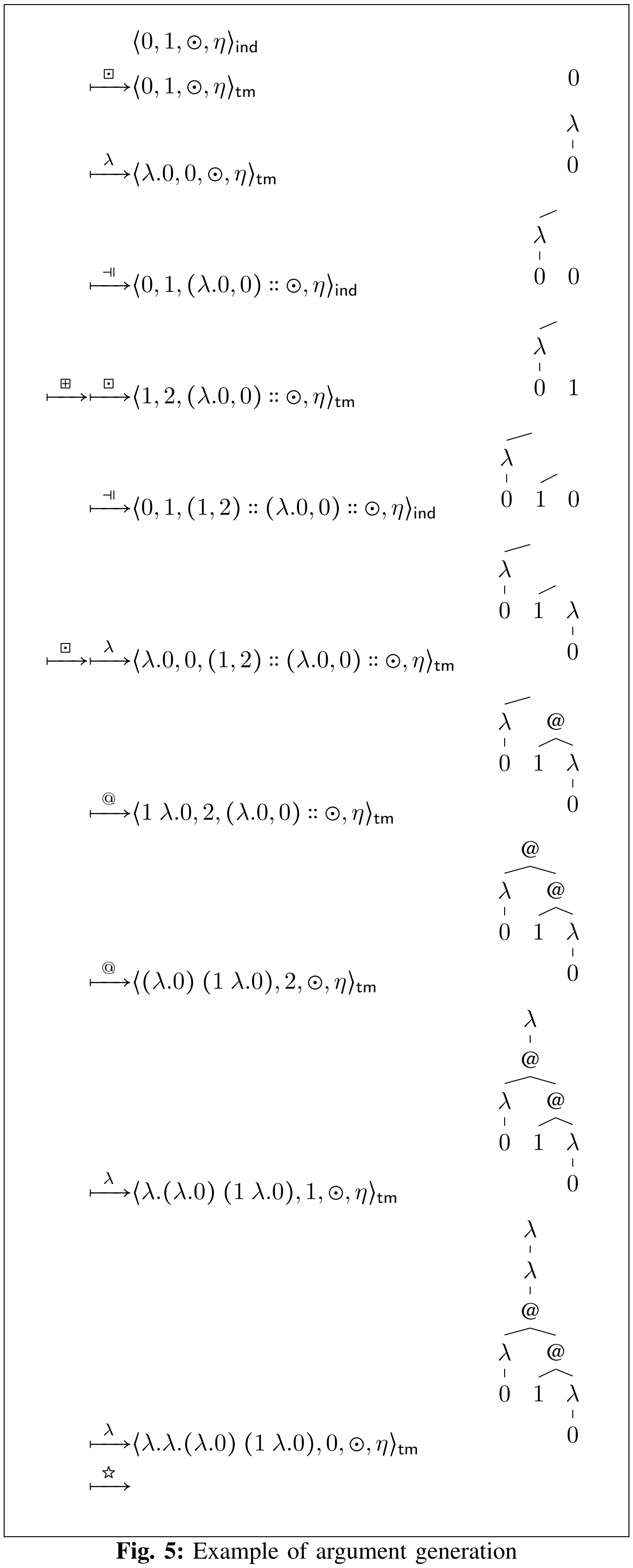
Lemma 20. *If t' is closed, then $\langle 0,1,\odot,(t,e)\rangle_{\mathsf{ind}} \xmapsto{\mathsf{Seq}(t')} \langle t,(t',\epsilon)::e,[]\rangle_{\mathsf{ev}}$.*

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

Theorem 21. $(t,e) \approx_{\mathsf{app}} (s,d)$ *iff* $\langle t,e,[]\rangle_{\mathsf{ev}} \approx_{\mathsf{m}} \langle s,d,[]\rangle_{\mathsf{ev}}$.

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

$$\{((t,e),(s,d)) \;\big|\; \langle t,e,[]\rangle_{\mathsf{ev}} \approx_{\mathsf{m}} \langle s,d,[]\rangle_{\mathsf{ev}}\}$$



$$\langle 0, 1, \odot, \eta\rangle_{\mathsf{ind}}$$
$$\xmapsto{\boxdot} \langle 0, 1, \odot, \eta\rangle_{\mathsf{tm}}$$
$$\xmapsto{\lambda} \langle \lambda.0, 0, \odot, \eta\rangle_{\mathsf{tm}}$$
$$\xmapsto{\dashv\!|} \langle 0, 1, (\lambda.0, 0) :: \odot, \eta\rangle_{\mathsf{ind}}$$
$$\xmapsto{\boxplus}\xmapsto{\boxdot} \langle 1, 2, (\lambda.0, 0) :: \odot, \eta\rangle_{\mathsf{tm}}$$
$$\xmapsto{\dashv\!|} \langle 0, 1, (1, 2) :: (\lambda.0, 0) :: \odot, \eta\rangle_{\mathsf{ind}}$$
$$\xmapsto{\boxdot}\xmapsto{\lambda} \langle \lambda.0, 0, (1, 2) :: (\lambda.0, 0) :: \odot, \eta\rangle_{\mathsf{tm}}$$
$$\xmapsto{@} \langle 1\ \lambda.0, 2, (\lambda.0, 0) :: \odot, \eta\rangle_{\mathsf{tm}}$$
$$\xmapsto{@} \langle (\lambda.0)\ (1\ \lambda.0), 2, \odot, \eta\rangle_{\mathsf{tm}}$$
$$\xmapsto{\lambda} \langle \lambda.(\lambda.0)\ (1\ \lambda.0), 1, \odot, \eta\rangle_{\mathsf{tm}}$$
$$\xmapsto{\lambda} \langle \lambda.\lambda.(\lambda.0)\ (1\ \lambda.0), 0, \odot, \eta\rangle_{\mathsf{tm}}$$
$$\xmapsto{\star}$$

Fig. 5: Example of argument generation

is an applicative bisimulation. Roughly, (t, e) evaluating to $(\lambda.t', e')$ means that (s, d) evaluates to $(\lambda.s', d')$ thanks to $\circledcirc$. Then for all closed t'', the sequence $\langle 0, 1, \odot, (t', e')\rangle_{\mathsf{ind}} \xmapsto{\mathsf{Seq}(t'')} \langle t', (t'', \epsilon) :: e', []\rangle_{\mathsf{ev}}$ can only be matched by $\langle 0, 1, \odot, (s', d')\rangle_{\mathsf{ind}} \xmapsto{\mathsf{Seq}(t'')} \langle s', (t'', \epsilon) :: d', []\rangle_{\mathsf{ev}}$, so we can conclude.

For the reverse implication, we show that

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

is a machine bisimulation, which is easy to check. □

C. Translation into HOcore

Figure 6 gives the translation of the AB machine in HOcore. We detail each component, starting with the evaluation mode, i.e., the KAM. We follow the same principles as in Section III: 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. A closure is represented by two messages, one containing the term on η_1 and one containing 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 explained later.

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 $n > 0$ go 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, 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 $[\![[]]\!] \parallel hd_{\mathsf{c}}(z).([\![t]\!] \parallel \overline{env}\langle \overline{hd_{\mathsf{e}}}\langle z\rangle \parallel \overline{env}\langle [\![e]\!]\rangle\rangle)$, so $[\![[]]\!]$ has to start the argument generating process, and the result has then to be sent on hd_{c} for the evaluation to restart. We write $\mathsf{Stuck}((t, e))$ for the process in parallel with $[\![[]]\!]$, which remains stuck during the whole generation process. We now explain how $\langle n, \kappa, \rho, \eta\rangle_{\mathsf{ind}}$ and $\langle t, \kappa, \rho, \eta\rangle_{\mathsf{tm}}$ are encoded, starting with κ and ρ.

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), we send on name $zero$ the expected behavior if $\kappa = 0$, and on $succ$ what to do otherwise. The translation of the counter receives both messages, executes the corresponding

1) Evaluation mode

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

$$\llbracket[\,]\rrbracket \triangleq \circledcirc(\_).(\overline{ind}\langle\llbracket 0\rrbracket\rangle \parallel \overline{k}\langle\llbracket 1\rrbracket_{\mathsf{c}}\rangle \parallel \overline{r}\langle\llbracket\odot\rrbracket\rangle \parallel \overline{initInd}\langle\mathbf{0}\rangle) \qquad \llbracket\epsilon\rrbracket \triangleq \mathbf{0}$$

$$\llbracket\eta :: \pi\rrbracket \triangleq \overline{hd_{\mathsf{c}}}\langle\llbracket\eta\rrbracket\rangle \parallel \overline{c}\langle\llbracket\pi\rrbracket\rangle \qquad \llbracket\eta :: e\rrbracket \triangleq \overline{hd_{\mathsf{e}}}\langle\llbracket\eta\rrbracket\rangle \parallel \overline{env}\langle\llbracket e\rrbracket\rangle$$

$$\llbracket t\ s\rrbracket \triangleq \mathsf{App}_{\mathsf{eval}}(\llbracket t\rrbracket,\llbracket s\rrbracket) \qquad \mathsf{App}_{\mathsf{eval}}(P_t,P_s) \triangleq c(x).env(y).\left(P_t \parallel \overline{c}\left\langle\overline{hd_{\mathsf{c}}}\left\langle\overline{\eta_1}\langle P_s\rangle \parallel \overline{\eta_2}\langle y\rangle\right\rangle \parallel x\right\rangle \parallel \overline{env}\langle y\rangle\right)$$

$$\llbracket\lambda.t\rrbracket \triangleq \mathsf{Lam}_{\mathsf{eval}}(\llbracket t\rrbracket) \qquad \mathsf{Lam}_{\mathsf{eval}}(P_t) \triangleq c(x).\left(x \parallel hd_{\mathsf{c}}(y).env(z).\left(P_t \parallel \overline{env}\left\langle\overline{hd_{\mathsf{e}}}\langle y\rangle \parallel \overline{env}\langle z\rangle\right\rangle\right)\right)$$

$$\llbracket n+1\rrbracket \triangleq \mathsf{Ind}_{\mathsf{eval}}(\llbracket n\rrbracket) \qquad \mathsf{Ind}_{\mathsf{eval}}(P_n) \triangleq env(x).(x \parallel hd_{\mathsf{e}}(\_).P_n)$$

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

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

2) Counter κ, stack ρ, and stuck process $\mathsf{Stuck}(\eta)$

$$\llbracket 0\rrbracket_{\mathsf{c}} \triangleq zero(x).succ(\_).x \qquad \llbracket\odot\rrbracket \triangleq mt(x).cons(\_).x$$

$$\llbracket\kappa+1\rrbracket_{\mathsf{c}} \triangleq \mathsf{Suk}(\llbracket\kappa\rrbracket_{\mathsf{c}}) \qquad \llbracket(t,\kappa)::\rho\rrbracket \triangleq \mathsf{ConsR}(\llbracket(t,\kappa)\rrbracket,\llbracket\rho\rrbracket)$$

$$\mathsf{Suk}(P_\kappa) \triangleq \overline{suk}\langle P_\kappa\rangle \parallel zero(\_).succ(x).x \qquad \mathsf{ConsR}(P_{hd},P_\rho) \triangleq \overline{hd_{\mathsf{r}}}\langle P_{hd}\rangle \parallel \overline{r}\langle P_\rho\rangle \parallel mt(\_).cons(x).x$$

$$\mathsf{Stuck}((t,e)) \triangleq hd_{\mathsf{c}}(z).\left(\llbracket t\rrbracket \parallel \overline{env}\left\langle\overline{hd_{\mathsf{e}}}\langle z\rangle \parallel \overline{env}\langle\llbracket e\rrbracket\rangle\right\rangle\right) \qquad \llbracket(t,\kappa)\rrbracket \triangleq \overline{w_1}\langle\llbracket t\rrbracket\rangle \parallel \overline{w_2}\langle\llbracket\kappa\rrbracket_{\mathsf{c}}\rangle$$

3) Creation of indices

$$\llbracket\langle n,\kappa,\rho,\eta\rangle_{\mathsf{ind}}\rrbracket \triangleq \overline{ind}\langle\llbracket n\rrbracket\rangle \parallel \overline{k}\langle\llbracket\kappa\rrbracket_{\mathsf{c}}\rangle \parallel \overline{r}\langle\llbracket\rho\rrbracket\rangle \parallel P_{\mathsf{rec}} \parallel \overline{initInd}\langle\mathbf{0}\rangle \parallel \mathsf{Stuck}(\eta)$$

$$\mathsf{RecInd} \triangleq initInd(\_).recind(x).(x \parallel \overline{recind}\langle x\rangle \parallel \mathsf{Succ} + \mathsf{Var})$$

$$\mathsf{Succ} \triangleq \boxplus(\_).ind(x).k(y).(\overline{ind}\langle\mathsf{Ind}_{\mathsf{eval}}(x)\rangle \parallel \overline{k}\langle\mathsf{Suk}(y)\rangle \parallel \overline{initInd}\langle\mathbf{0}\rangle)$$

$$\mathsf{Var} \triangleq \boxdot(\_).ind(x).(\overline{tm}\langle x\rangle \parallel \overline{initTm}\langle\mathbf{0}\rangle)$$

4) Creation of terms

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

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

$$\mathsf{AppFun}(P_\kappa,P_\rho) =\!\dashv\!|\,(\_).tm(x).\left(\overline{r}\left\langle\mathsf{ConsR}\left(\overline{hd_{\mathsf{r}}}\left\langle\overline{w_1}\langle x\rangle \parallel \overline{w_2}\langle P_\kappa\rangle\right\rangle, P_\rho\right)\right\rangle \parallel \overline{ind}\langle\llbracket 0\rrbracket\rangle \parallel \overline{k}\langle\llbracket 1\rrbracket_{\mathsf{c}}\rangle \parallel \overline{initInd}\langle\mathbf{0}\rangle\right)$$

$$\mathsf{Done} \triangleq \text{☆}(\_).tm(x).\left(\overline{hd_{\mathsf{c}}}\left\langle\overline{\eta_1}\langle x\rangle \parallel \overline{\eta_2}\langle\llbracket\epsilon\rrbracket\rangle\right\rangle \parallel \overline{c}\langle\llbracket[\,]\rrbracket\rangle\right)$$

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

$$\mathsf{RecMax} \triangleq init1(x_1).init2(x_2).recmax(y).(y \parallel \overline{recmax}\langle y\rangle \parallel \mathsf{Max}(x_1,x_2))$$

$$\mathsf{Max}(P_1,P_2) \triangleq P_1 \parallel \overline{zero}\left\langle max2(x).\overline{resu}\langle x\rangle\right\rangle \parallel \overline{succ}\left\langle suk(x_1).\begin{pmatrix}P_2 \parallel \overline{zero}\left\langle max1(x).\overline{resu}\langle x\rangle\right\rangle \\ \parallel \overline{succ}\left\langle suk(x_2).(\overline{init1}\langle x_1\rangle \parallel \overline{init2}\langle x_2\rangle)\right\rangle\end{pmatrix}\right\rangle$$

$$\mathsf{RecTm} \triangleq initTm(\_).rectm(x).\begin{pmatrix} x \parallel \overline{rectm}\langle x\rangle \parallel r(y).y \parallel \overline{cons}\left\langle\begin{matrix}k(z).hd_{\mathsf{r}}(y_1).r(y_2). \\ (\mathsf{Lambda}(z) \parallel y) + \mathsf{AppFun}(z,y) + \mathsf{App}(z,y_1,y_2)\end{matrix}\right\rangle \\ \parallel \overline{mt}\left\langle k(z).\begin{pmatrix} z \parallel \overline{zero}\left\langle\begin{matrix}(\mathsf{Lambda}(z) \parallel \llbracket\odot\rrbracket) \\ + \mathsf{AppFun}(z,\llbracket\odot\rrbracket) + \mathsf{Done}\end{matrix}\right\rangle \\ \parallel \overline{succ}\left\langle suk(\_).\begin{matrix}(\mathsf{Lambda}(z) \parallel \llbracket\odot\rrbracket) \\ + \mathsf{AppFun}(z,\llbracket\odot\rrbracket)\end{matrix}\right\rangle\end{pmatrix}\right\rangle\end{pmatrix}$$

Fig. 6: Translation of the AB machine

one (e.g., the one on $zero$ for the encoding of 0), and discards the other. Apart from that, κ is translated as a natural number. 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.

After flagging $\circledcirc$, the process $[\![[]]\!]$ starts the argument generation process in index mode: the index being built is sent on ind (here, initialized with $[\![0]\!]$), the counter on k, and the stack on r. 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 $\mathsf{Ind}_{\mathsf{eval}}$) and the counter (with Suk), and relaunches the RecInd process with a message on $initInt$. 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 process RecTm.

The goal of RecTm is to non-deterministically choose between the four transitions available in term mode, namely $\xmapsto{\lambda}$, $\xmapsto{\dashv\shortmid}$, $\xmapsto{@}$, and $\xmapsto{\star}$. However, some of these transitions have requirements: $\xmapsto{@}$ needs $\rho \neq \odot$ and $\xmapsto{\star}$ 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 $cons$, which makes a choice between λ, $\dashv\shortmid$, 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, $\dashv\shortmid$, or $\star$ (represented by Done), otherwise, only λ or $\dashv\shortmid$ are possible.

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.

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.

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.

$$
\begin{array}{rcl}
\langle n, \kappa, \rho \rangle_{\mathsf{ind}} \xmapsto{\boxplus} \langle n+1, \kappa+1, \rho \rangle_{\mathsf{ind}} & & (\textsc{Suc}) \\
\langle n, \kappa, \rho \rangle_{\mathsf{ind}} \xmapsto{\boxdot} \langle n, \kappa, \rho \rangle_{\mathsf{tm}} & & (\textsc{Var}) \\
\langle t, \kappa+1, \rho \rangle_{\mathsf{tm}} \xmapsto{\lambda} \langle \lambda.t, \kappa, \rho \rangle_{\mathsf{tm}} & & (\textsc{Lambda}) \\
\langle t, 0, \rho \rangle_{\mathsf{tm}} \xmapsto{\lambda} \langle \lambda.t, 0, \rho \rangle_{\mathsf{tm}} & & (\textsc{Lambda0}) \\
\langle t, \kappa, \rho \rangle_{\mathsf{tm}} \xmapsto{\dashv\shortmid} \langle 0, 1, (t, \kappa) :: \rho \rangle_{\mathsf{ind}} & & (\textsc{AppPush}) \\
\langle t, \kappa_1, (s, \kappa_2) :: \rho \rangle_{\mathsf{tm}} \xmapsto{\overleftarrow{@}} \langle t\, s, \mathsf{max}(\kappa_1, \kappa_2), \rho \rangle_{\mathsf{tm}} & & (\overleftarrow{\textsc{App}}) \\
\langle t, \kappa_1, (s, \kappa_2) :: \rho \rangle_{\mathsf{tm}} \xmapsto{\overrightarrow{@}} \langle s\, t, \mathsf{max}(\kappa_1, \kappa_2), \rho \rangle_{\mathsf{tm}} & & (\overrightarrow{\textsc{App}}) \\
\langle t, 0, \odot \rangle_{\mathsf{tm}} \xmapsto{\circledast} \langle t, \epsilon, [\,] \rangle_{\mathsf{ev}} & & (\textsc{Start}) \\
\\
\langle \lambda.t, e, [\,] \rangle_{\mathsf{ev}} \xmapsto{\star} & & (\textsc{Done})
\end{array}
$$

Fig. 7: Contextual equivalence machine

Full abstraction: We use the notions of Section IV-D to prove the correspondence between the AB machine and its translation. The definition of next carries over to the translation of the AB machine, and Lemmas 12 and 13 still hold (but without terminating transitions). Recall that $\approx_{\mathsf{HO}}$ stands for $\approx_{\mathsf{HO}}^{\mathbb{H}}$, where $\mathbb{H}$ contains the names of the translation that are not flags. The proof of the following theorem is then the same as for Theorem 14.

Theorem 22. $C \approx_{\mathsf{m}} C'$ *iff* $[\![C]\!] \approx_{\mathsf{HO}} [\![C']\!]$.

We then deduce a full abstraction result between λ-calculus with applicative bisimilarity and HOcore.

Corollary 23. *If* (t, e) *and* (s, d) *are closed closures, then* $(t, e) \approx_{\mathsf{app}} (s, d)$ *iff* $[\![\langle t, e, [\,] \rangle_{\mathsf{ev}}]\!] \approx_{\mathsf{HO}} [\![\langle s, d, [\,] \rangle_{\mathsf{ev}}]\!]$.

D. Internalizing Contextual Equivalence

Corollary 23 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 this result directly, by internalizing contextual equivalence in an abstract machine.

Figure 7 gives the transitions of this machine, except for the $\xmapsto{\tau}$ transitions, which are the same as in Section V-A. 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 $\xmapsto{\circledast}$ 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 $\star$ 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 V-B). 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 $s\ t$.

Example 24. *We show how to generate the context* $\lambda.(0\,0)\ (\Box\,0)$ *around* t.

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

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

Theorem 25. *If* t *and* s *are closed terms, then* $t \approx_{\mathsf{ctx}} s$ *iff* $[\![\langle t, 0, \odot\rangle_{\mathsf{tm}}]\!] \approx_{\mathsf{HO}} [\![\langle s, 0, \odot\rangle_{\mathsf{tm}}]\!]$.

VI. Conclusion and Future Work

We propose encodings of the call-by-name λ-calculus 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 the call-by-name λ-calculus.

Because we rely on abstract machines, our encodings are not compositional. 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 a LTS (a flags-generating machine), which is closer to the HOcore equivalence. We believe this internalization technique can be applied to other languages for which an abstract machine has been defined, like, e.g., the call-by-value calculus and its CK machine [12] (see [8, Appendix C]), or a calculus with control operators [7]. Even though the bisimilarities for these calculi can be quite intricate, it should be always possible to generate a context as in Section V-D to internalize contextual equivalence.

The encodings of the extended abstract machines into HOcore rely on the same principles, e.g., to represent stacks, nondeterministic 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 is deterministic (up to flags for choice processes, as in Lemma 12) and with an operational correspondence result similar to Lemma 13. As these two ingredients are almost sufficient to get full abstraction between machines and HOcore, it would give us Theorem 14 or Theorem 22 for free.

Finally, we aim at translating the full λ-calculus, without fixing a reduction strategy. Such an encoding is proposed in [9], 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.

Acknowledgements: This work has been partially supported by PHC Polonium and by National Science Centre, Poland, grant no. 2014/15/B/ST6/00619. Part of this work has been carried out while Sergueï Lenglet was at the Irisa lab, Université de Rennes 1, thanks to a CNRS funding. Damien Pous is funded by the European Research Council (ERC) under the European Unions Horizon 2020 programme (CoVeCe, grant agreement No 678157).

References

[1] S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.
[2] B. Accattoli. Evaluating functions as processes. In *TERMGRAPH 2013*, volume 110 of *EPTCS*, pages 41–55, 2013.
[3] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP'03*, pages 8–19, 2003. ACM Press.
[4] R. M. Amadio. A decompilation of the pi-calculus and its application to termination. *CoRR*, abs/1102.2339, 2011.
[5] E. Beffara. Functions as proofs as processes. *CoRR*, abs/1107.4160, 2011.
[6] M. Berger, K. Honda, and N. Yoshida. Sequentiality and the pi-calculus. In *TLCA 2001*, number 2044 in *LNCS*, pages 29–45, 2001. Springer.
[7] M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *LMCS*, 1(2:5):1–39, 2005.
[8] M. Biernacka, D. Biernacki, S. Lenglet, P. Polesiuk, D. Pous, and A. Schmitt. Fully abstract encodings of λ-calculus in HOcore through abstract machines Research Report RR-9052, Inria, 2017. Available at http://hal.inria.fr/hal-01507625.
[9] X. Cai and Y. Fu. The λ-calculus in the π-calculus. *Mathematical Structures in Computer Science*, 21(5):943–996, 2011.
[10] M. Cimini, C. S. Coen, and D. Sangiorgi. Functions as processes: Termination and the $\overline{\lambda}\mu\tilde{\mu}$-calculus. In *TGC 2010*, volume 6084 of *LNCS*, pages 73–86, 2010. Springer.
[11] P. Downen, L. Maurer, Z. M. Ariola, and D. Varacca. Continuations, processes, and sharing. In *PPDP 2014*, pages 69–80, 2014. ACM.
[12] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.
[13] K. Honda, N. Yoshida, and M. Berger. Process types as a descriptive tool for interaction - control and the pi-calculus. In *RTA-TLCA 2014*, volume 8560 of *LNCS*, pages 1–20, 2014. Springer.
[14] J.-L. Krivine. A call-by-name lambda-calculus machine. *HOSC*, 20(3):199–207, 2007.
[15] I. Lanese, J. A. Pérez, D. Sangiorgi, and A. Schmitt. On the expressiveness of polyadic and synchronous communication in higher-order process calculi. In *ICALP 2010*, volume 6199 of *LNCS*, pages 442–453, 2010. Springer.
[16] I. Lanese, J. A. Pérez, D. Sangiorgi, and A. Schmitt. On the expressiveness and decidability of higher-order process calculi. *Information and Computation*, 209(2):198–226, 2011.
[17] S. B. Lassen. Bisimulation in untyped lambda calculus: Böhm trees and bisimulation up to context. In *MFPS 1999*, volume 20 of *ENTCS*, pages 346–374, New Orleans, LA, Apr. 1999.
[18] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
[19] D. Sangiorgi. Bisimulation for higher-order process calculi. *Information and Computation*, 131(2):141–178, 1996.
[20] D. Sangiorgi. From lambda to pi; or, rediscovering continuations. *Mathematical Structures in Computer Science*, 9(4):367–401, 1999.
[21] D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
[22] B. Toninho, L. Caires, and F. Pfenning. Functions as session-typed processes. In *FOSSACS'12*, volume 7213 of *LNCS*, pages 346–360, 2012.
[23] S. van Bakel and M. G. Vigliotti. A fully-abstract semantics of lambda-mu in the pi-calculus. In *CL&C 2014*, volume 164 of *EPTCS*, pages 33–47, 2014.
[24] N. Yoshida, M. Berger, and K. Honda. Strong normalisation in the pi -calculus. *Information and Computation*, 191(2):145–202, 2004.