Lambda and pi calculi, CAM and SECD machines

Vasco Thudichum Vasconcelos
Department of Informatics
Faculty of Sciences
University of Lisbon

February 2001
(revised November 2002, August 2003)

Abstract

We analyse machines that implement the call-by-value reduction strategy of the λ-calculus: two environment machines—CAM and SECD—and two encodings into the π-calculus—due to Milner and Vasconcelos. To establish the relation between the various machines, we setup a notion of reduction machine and two notions of correspondences: operational—in which a reduction step in the source machine is mimicked by a sequence of steps in the target machine—and convergent—where only reduction to normal form is simulated. We show that there are operational correspondences from the λ-calculus into CAM, and from CAM and from SECD into the π-calculus. Plotkin completes the picture by showing that there is a convergent correspondence from the λ-calculus into SECD.

1 Introduction

The call-by-value reduction strategy of the λ-calculus equips a large number of today's programming languages. To study the operational aspects of this reduction strategy Plotkin used Landin's SECD, a stack-based, environment machine [6, 11]. Based on the Categorical Abstract Machine and inspired by the Krivine's lazy abstract machine, Curien proposed a different environment machine, which we call CAM throughout this paper [4]. From a different community, two interpretations of the call-by-value λ-calculus into the π-calculus are known: one proposed by Milner, and further studied by Pierce and Sangiorgi [7, 8, 9], the other proposed by the author, but lacking, to date a systematic study [13].

In his paper on encodings of the λ-calculus into the π-calculus, Milner uses the term 'environment entry' to describe a process simulating the behaviour of some function on a given channel [7, 8]. The term comes probably from the so called environment machines, devices that describe implementations of the λ-calculus, capable of efficiently performing substitutions by maintaining a map from variables to function closures (the environment). The natural question arises as "what is the relation between the π-encodings of the λ-calculus and the environment machines?". We answer the question by showing that there is a close correspondence (indeed, an operational correspondence) between the CAM machine and Milner's encoding, and between the SECD machine and Vasconcelos' encoding. Also, because there is an operational correspondence from the λ-calculus into CAM, but no such correspondence into SECD, we learn that the two encodings are quite different. In fact, our study reveals:

1. How close the π-encodings are to the environment machines: Milner's encoding mimics step-by-step the CAM machine; Vasconcelos' does the same for the SECD machine;

2. The inherent difference between the two encodings: Milner's follows the reduction of a λ-term step-by-step; Vasconcelos' cannot follow intermediate steps.

In order to establish the correspondences between the various machines, we setup a notion of *reduction machine*—composed of a set of states, a reduction relation, and an equivalence relation, satisfying some simple conditions—and a notion of *correspondence*—a partial function on states that preserves equivalence. We identify two kinds of correspondences: *operational* where a reduction step in the source machine is mimicked by a sequence of steps in the target machine, modulo equivalence; and *convergent* where reduction to normal form in the source machine is mimicked by reduction to normal form in the target machine, again modulo equivalence. Reduction machines and correspondences are then studied on their own, paving way for the results on the particular correspondences studied in the paper.

Equipped with the notion of reduction machines we set up five concrete machines: the usual λ-calculus with the call-by-value reduction strategy [11], the CAM machine [4], the SECD machine [11], and two π-calculus based machines. For the first π-based machine, we pick processes of the asynchronous π-calculus [3, 5] typable under the input/output type system, together with deterministic reduction, and strong barbed congruence [9].

For the second π-based machine, we pick contexts for states. A notion of reduction and equivalence for π-calculus contexts is then defined. A (typed) context performs a (deterministic) reduction step if the (typed) process in the hole plays no rôle in reduction, that is, if the reduction step of the filled-in context is independent of the process that fills the hole. For the equivalence, we isolate a subset of names that cannot be free in the hole. Then, we play the same game as for reduction: two contexts are strong barbed congruent if the processes obtained by filling the hole with the same process are strong barbed congruent. In our particular case, the set of names that cannot be free in the hole corresponds to the λ-variables (which form a subset of the π-names).

We then study four correspondences: two from the call-by-value λ-calculus into the environment machines (CAM and SECD), another two from each environment machine into a π-based machine. The encoding of the CAM machine into the π-calculus is based on a variant of Milner's encoding [7, 9, 12], obtained by partially evaluating applications whose left operand (the function) is already a value. The encoding of the SECD machine into π-calculus contexts is based on Vasconcelos encoding [13, 14]. The advantage of proceeding through environment machines is that proofs of the operational correspondences are quite simple: in each case it resumes to a simple analysis of the reduction rules, appealing directly to the Replication Theorems [9, 12].

We also study direct interpretations of the λ-calculus into the π-calculus, and show that they are sound. The direct encodings allow to quantify the number of π-reduction steps needed to mimic a λ-step: exactly two for the variant of the Milner's encoding (but not for the original version), and two (in average, for one can only compare values) in the case of the context encoding.

As mentioned above, the two encodings are quite different, for one reflects the CAM machine, while the other the SECD machine. The process-machine mimics, one-by-one, the λ-steps of a term (theorem 32):

$$M \rightarrow_{\mathsf{v}} N \text{ implies } [\![M]\!] \rightarrow^2_{\mathsf{d}}\simeq [\![N]\!].$$

The context translation mimics λ-reduction to a value (theorem 34),

$$M \downarrow^n_{\mathsf{v}} N \text{ implies } [\![M]\!] \downarrow^{2n}_{\mathsf{d}}\simeq [\![N]\!].$$

but cannot follow the intermediate steps:

$$(\lambda x \mathbf{I} x)V \rightarrow_{\mathsf{v}} \mathbf{I}V, \text{ but } [\![(\lambda x \mathbf{I} x)V]\!] \not\rightarrow^*_{\mathsf{d}}\simeq [\![\mathbf{I}V]\!].$$

The big picture. The five machines studied in this paper are Lambda (section 3.1), CAM (section 3.2), SECD (section 3.3), Process (section 3.5), and Context (section 3.6). For M_1, M_2 two machines, write $M_1 \to M_2$ if there is a correspondence from M_1 to M_2. For the *operational* case, we have the following scenario.

$$\begin{tikzcd}
 & \mathsf{Lambda} \arrow[dl, "\text{Theorem 18}"'] \arrow[ddl, "\text{Theorem 31.1}"] & \\
\mathsf{CAM} \arrow[d, "\text{Theorem 24}"'] & & \mathsf{SECD} \arrow[d, "\text{Theorem 29}"] \\
\mathsf{Process} & & \mathsf{Context}
\end{tikzcd}$$

The Lambda to CAM to Process operational correspondences are the object of sections 4.1, and 4.3. Then, we may obtain the Lambda to Process correspondence by composition (section 4.5). In a different partition of the diagram, the SECD to Context machine is the object of section 4.4.

For the *convergent* case, Plotkin provides the missing link, thus completing the puzzle.

$$\begin{tikzcd}
 & \mathsf{Lambda} \arrow[dl, "\text{Corollary 19}"'] \arrow[ddl, "\text{T 31.3}"] \arrow[dr, "\text{Plotkin [11]}"] \arrow[ddr, "\text{T 31.2}"'] & \\
\mathsf{CAM} \arrow[d, "\text{Corollary 25}"'] & & \mathsf{SECD} \arrow[d, "\text{Corollary 30}"] \\
\mathsf{Process} & & \mathsf{Context}
\end{tikzcd}$$

The arrows in the operational diagram are inherited, since operational correspondences are also convergent (theorem 5). The Lambda to SECD correspondence is the object of section 4.2, the result is by Plotkin [11]. Then, we bridge the Lambda and the Context machines via composition (section 4.5).

Outline. The next section introduces the notions of reduction machine and of correspondence, together with basic results on these. Section 3 introduces the five machines mentioned above, and section 4 studies the correspondences. Section 5 concludes the paper.

2 Correspondences

This section introduces the notion of reduction machine, and two kinds of correspondences between them. *Reduction machines* are given by a set of states, a transition relation on states, and an equivalence relation also on states.

Definition 1 (Reduction machine). *A* reduction machine *is a triple $\langle S, \rightarrow, \sim\rangle$, where S is a set (the* set of states*), $\rightarrow$ is a relation in $S \times S$ (the* reduction relation*), and $\sim$ is an equivalence relation in $S \times S$, such that*

1. *$s \rightarrow s'$, $s \rightarrow s''$ implies $s' \sim s''$, and*

2. *the relation $\sim\rightarrow$ is contained in the relation $\rightarrow^*\sim$.*

Notice that we do not require that $\sim$ and $\rightarrow$ commute, not even that $\sim\rightarrow$ is contained in $\rightarrow\sim$. Both the CAM and the SECD reduction machines defined in sections 3.2, 3.3 do not satisfy the latter condition. Also, a simple induction on the length of the reduction shows that $\sim\rightarrow^*$ is contained in $\rightarrow^*\sim$ as well.

For a given machine $\langle S, \rightarrow, \sim\rangle$, we say that: a) state s *reduces*, and write $s \rightarrow$, to mean $s \rightarrow s'$ for some s'; b) state s *reduces in n steps* to state s' ($n \geq 0$), and write $s \rightarrow^n s'$, if $s \rightarrow s_1 \rightarrow \ldots \rightarrow s_n = s'$; when n is not important we write $s \rightarrow^* s'$; c) state s *converges* in n steps to s' ($n \geq 0$), and write $s \downarrow^n s'$, if $s \rightarrow^n s' \not\rightarrow$; when n is not important, we write $s \downarrow s'$; and when s' is also not important, we write $s \downarrow$.

Correspondences are functions from machine states into machine states that preserve state equivalence. We distinguish two kinds of correspondences.

Definition 2 (Correspondences). *Given two reduction machines $\langle S, \rightarrow_S, \sim_S\rangle$, $\langle R, \rightarrow_R, \sim_R\rangle$, and a partial function* $\mathsf{load} : S \rightarrow R$*, we say that*

1. load *is a* correspondence *if $s \sim s'$ implies $\mathsf{load}(s) \sim \mathsf{load}(s')$;*

2. load *is* operational *if it is a correspondence, and*

   (a) $s \rightarrow s'$ implies $\mathsf{load}(s) \rightarrow^*\sim \mathsf{load}(s')$;

   (b) $\mathsf{load}(s) \downarrow$ implies $s \downarrow$; and

   (c) $\mathsf{load}(s) \rightarrow$ implies $s \rightarrow$.

3. load *is* convergent *if it is a correspondence, and*

   (a) $s \downarrow s'$ implies $\mathsf{load}(s) \downarrow\sim \mathsf{load}(s')$; and

   (b) $\mathsf{load}(s) \downarrow$ implies $s \downarrow$.

Operational correspondences are not adequate, but, given a mild condition on the target machine, the correspondences become adequate.

Proposition 3. *Given two reduction machines $\langle S, \rightarrow_S, \sim_S\rangle$, $\langle R, \rightarrow_R, \sim_R\rangle$ such that $r \sim_R r' \not\rightarrow_R$ implies $r \not\rightarrow_R$, and an operational correspondence* $\mathsf{load} : S \rightarrow R$*,*

1. *$s \rightarrow^* s'$ implies $\mathsf{load}(s) \rightarrow^* \sim \mathsf{load}(s')$;*

2. *$s \downarrow s'$ implies $\mathsf{load}(s) \downarrow \sim \mathsf{load}(s')$;*

3. (Adequacy) *$s \downarrow$ iff $\mathsf{load}(s) \downarrow$.*

Proof. 1. Induction on the length of reduction. When $s = s'$, we know that $\mathsf{load}(s) = \mathsf{load}(s')$. Otherwise suppose that $s \rightarrow^* s' \rightarrow s''$. By induction we know that $\mathsf{load}(s) \rightarrow^* \sim \mathsf{load}(s')$, and from the definition of operational correspondence, that $\mathsf{load}(s') \rightarrow^* \sim \mathsf{load}(s'')$. Conclude with the fact that $\sim \rightarrow^*$ is contained in $\rightarrow^* \sim$. *2.* Use the above clause and the contrapositive of 2.2c to obtain $\mathsf{load}(s) \rightarrow^* r \sim \mathsf{load}(s')$, for some r. Use the hypothesis to conclude that $r \not\rightarrow$. *3.* Clause above and definition 2.2b. $\square$

Adequacy in the convergence scenario is easier to establish.

Proposition 4. *Given two reduction machines $\langle S, \rightarrow_S, \sim_S \rangle$, $\langle R, \rightarrow_R, \sim_R \rangle$, and a convergence correspondence $\mathsf{load} : S \rightarrow R$,*

1. *$\mathsf{load}(s) \downarrow r$ implies $s \downarrow s'$ and $r \sim \mathsf{load}(s')$;*

2. *$s \downarrow$ implies $\mathsf{load}(s) \downarrow$;*

3. (Adequacy) *$s \downarrow$ iff $\mathsf{load}(s) \downarrow$.*

Proof. 1, 2. Directly from the definition. *3.* Definition and clause 2. $\square$

The main result concerning correspondences says that operational correspondences are also convergence correspondences, and that the composition of correspondences is also a correspondence.

Theorem 5. *1. Operational correspondences are convergent;*

2. The composition of two operational correspondences is operational;

3. The composition of two convergence correspondences is convergent;

4. The composition of a convergent with an operational correspondence is convergent.

Proof. 1. Proposition 3.2 and definition 2.2b. *2.* The second and the third clauses in the definition of operational correspondences are direct. For the first, let R, S, T be the three machines involved. Using proposition 3.1 on R, we have $\mathsf{load}(s) \rightarrow_R^* r \sim_R \mathsf{load}(s')$; using the same proposition on S, we have $\mathsf{load}(\mathsf{load}(s)) \rightarrow_T^* \sim_T \mathsf{load}(r)$. By definition of correspondence $\mathsf{load}(r) \sim_T \mathsf{load}(\mathsf{load}(s'))$; hence $\mathsf{load}(\mathsf{load}(s)) \rightarrow_T^* \sim_T \mathsf{load}(\mathsf{load}(s'))$, as required. *3.* Easy. *4.* Clauses 1 and 3 above. $\square$

States: term

$$\begin{array}{rrcl} \mathsf{variable} & u,v,w,x,y,z & & \\ \mathsf{value} & V & ::= & x \mid \lambda x M \\ \mathsf{term} & M & ::= & V \mid (MN) \end{array}$$

Reduction: $\rightarrow_{\mathsf{v}}$

$$\begin{array}{rcll} (\lambda x M)V & \rightarrow_{\mathsf{v}} & M\{V/x\} & (\beta) \\ VM & \rightarrow_{\mathsf{v}} & VM' \quad \text{if} \quad M \rightarrow_{\mathsf{v}} M' & (\nu) \\ MN & \rightarrow_{\mathsf{v}} & M'N \quad \text{if} \quad M \rightarrow_{\mathsf{v}} M' & (\mu) \end{array}$$

Equivalence: $\equiv_\alpha$

Figure 1: The call-by-value Lambda machine

3 Five reduction machines

This section presents the following machines, all taken from the literature: the λ-calculus equipped with the call-by-value strategy, the CAM machine, the SECD machine, and two machines based on the π-calculus.

3.1 The call-by-value machine

This section introduces Plotkin's call-by-value λ-calculus [11]. We presuppose a countable set $\mathsf{variable}$ of variables. The sets of values and terms are defined in figure 1. We say that a variable x occurs *free* in a term M is x is not in the scope of a λx; x occurs *bound* otherwise. The set $\text{fv}(M)$ of the *free variables* in M is defined accordingly, and so is the result of substituting N for the free occurrences of x in M, denoted by $M\{N/x\}$, as well as the *alpha equivalence*, denoted by $\equiv_\alpha$. A term M is closed if $\text{fv}(M) = \emptyset$. The set of closed λ-terms is denoted by term^0. The *reduction relation* over term, written $\rightarrow_{\mathsf{v}}$, is the smallest relation satisfying the rules in figure 1.

The $\rightarrow_{\mathsf{v}}$ relation is Plotkin's left reduction: "If $M \rightarrow_{\mathsf{v}} N$, then N is gotten from M by reducing the leftmost redex, not in the scope of a λ" [11]. Notice that, since states are closed terms, amounts to say that $M \downarrow^n N$ is saying that $M \rightarrow^n N$ and N is a value.

The reduction machine we are interested in operates on closed terms and is defined in figure 1.

Proposition 6. *The triple* $\langle \mathsf{term}^0, \rightarrow_{\mathsf{v}}, \equiv_\alpha \rangle$ *is a reduction machine.*

Proof. For the first clause in definition 1 we know that $s' = s''$. For the second, $\rightarrow_{\mathsf{v}}$ commutes with $\equiv_\alpha$. □

States: cam

$$
\begin{array}{rrcll}
\mathsf{environment} & e & ::= & \{x_1 := vc_1, \ldots, x_n := vc_n\} & n \geq 0, x_i \text{ distinct} \\
\mathsf{valueclosure} & vc & ::= & (\lambda x M)[e] \quad \text{if} \quad \mathrm{fv}(\lambda x M) \subseteq \mathrm{dom}(e) & \\
\mathsf{closure} & c & ::= & M[e] \quad \text{if} \quad \mathrm{fv}(M) \subseteq \mathrm{dom}(e) & \\
\mathsf{stack} & s & ::= & \mathsf{l} : vc : s \quad | \quad \mathsf{r} : c : s \quad | \quad \mathsf{nil} & \\
\mathsf{cam} & k & ::= & \langle c, s \rangle &
\end{array}
$$

Reduction: $\rightarrow_{\mathsf{k}}$

$$
\begin{array}{rcll}
\langle x[e], s\rangle & \rightarrow_{\mathsf{k}} & \langle e(x), s\rangle & (\textsc{Var}) \\
\langle (MN)[e], s\rangle & \rightarrow_{\mathsf{k}} & \langle M[e], \mathsf{r} : N[e] : s\rangle & (\textsc{App}) \\
\langle vc, \mathsf{r} : c : s\rangle & \rightarrow_{\mathsf{k}} & \langle c, \mathsf{l} : vc : s\rangle & (\textsc{Exch}) \\
\langle vc, \mathsf{l} : (\lambda x M)[e] : s\rangle & \rightarrow_{\mathsf{k}} & \langle M[e\{x := vc\}], s\rangle & (\textsc{Call})
\end{array}
$$

Equivalence: $\sim_{\mathsf{k}}$

$$
\begin{array}{rcl}
\langle (VN)[], s\rangle & \sim_{\mathsf{k}} & \langle N[], \mathsf{l} : V[] : s\rangle \\
\langle (MN)[], s\rangle & \sim_{\mathsf{k}} & \langle M[], \mathsf{r} : N[] : s\rangle \\
\langle M[x := V[]], s\rangle & \sim_{\mathsf{k}} & \langle (M\{V/x\})[], s\rangle
\end{array}
$$

Figure 2: The CAM reduction machine

Example 7. *Consider the term $(\lambda x \mathbf{I} x)V$, where V is a closed value, and $\mathbf{I}$ the λyy combinator. We have:*

$$(\lambda x \mathbf{I} x)V \rightarrow_{\mathsf{v}} \mathbf{I}V \rightarrow_{\mathsf{v}} V.$$

3.2 The CAM machine

This section introduces the CAM machine [4], following the presentation of Amadio and Curien [1]. We start with environments and closures, two notions mutually defined; $\mathsf{closure}$ is the set of closures, and $\mathsf{environment}$ the set of environments. An *environment* is a partial function of finite domain from variables into closures. A *closure* c is a pair $M[e]$ in $\mathsf{term} \times \mathsf{environment}$. We evaluate closures $M[e]$ such that $\mathrm{fv}(M) \subseteq \mathrm{dom}(e)$. Notice that an empty environment constitutes the base of the recursive definition: if M is a closed term, then $M[\emptyset]$ is a closure. *Environment update* is captured by the operation $e\{x := vc\}$, denoting the unique environment e' such that $e'(y) = e(y)$ if $y \neq x$, and $e'(x) = vc$ otherwise.

The CAM machine uses a stack to keep track of terms waiting to be evaluated or waiting for their arguments to be evaluated. In order to distinguish what is on top of the stack, the machine uses two markers, l and r, specifying whether the next term in the stack is the left side of an application (function) or the right side (argument). Elements in a stack are separated by a colon, the top of the stack is at the left; the empty stack is denoted by nil. We often omit the trailing $: \mathsf{nil}$ in a stack.

The CAM machine is defined in figure 2: the reduction function, written $\rightarrow_{\mathsf{k}}$, is the smallest relation satisfying the rules in the figure; the equivalence relation on states, written $\sim_{\mathsf{k}}$, is the smallest equivalence relation that contains the equalities in the same figure.

Given a closed term M, the machine starts with state $\langle M[\emptyset], \mathsf{nil}\rangle$. We can easily check that terminal states are of the form $\langle vc, \mathsf{nil}\rangle$.

Example 8. *Recall the term $(\lambda x \mathbf{I} x)V$ from example 7. We have:*

$$\begin{array}{rlr}
\langle (\lambda x \mathbf{I} x)V[], \mathsf{nil}\rangle & \rightarrow & (\textsc{App}) \\
\langle (\lambda x \mathbf{I} x)[], \mathsf{r} : V[]\rangle & \rightarrow & (\textsc{Exch}) \\
\langle V[], \mathsf{l} : (\lambda x \mathbf{I} x)[]\rangle & \rightarrow & (\textsc{Call}) \\
\langle (\mathbf{I} x)[x := V[]], \mathsf{nil}\rangle & \rightarrow^3 & (\textsc{App},\textsc{Exch},\textsc{Var}) \\
\langle V[][x := V[]], \mathsf{l} : I[x := V[]]\rangle & \rightarrow & (\textsc{Call}) \\
\langle y[y := V[], x := V[]], \mathsf{nil}\rangle & \rightarrow & (\textsc{Var}) \\
\langle V[], \mathsf{nil}\rangle & &
\end{array}$$

Notice that the state $\langle (\mathbf{I}x)[x := V[]], \mathsf{nil}\rangle$ above is equivalent to $\langle \mathbf{I}V[], \mathsf{nil}\rangle$, hence $\langle (\lambda x \mathbf{I} x)V[], \mathsf{nil}\rangle \rightarrow_{\mathsf{k}}^{*}\sim \langle \mathbf{I}V[], \mathsf{nil}\rangle \rightarrow_{\mathsf{k}}^{*} \langle V[], \mathsf{nil}\rangle$ (cf. example 7). In section 4.1 we show that there is an operational correspondence from Lambda *to* CAM.

Proposition 9. *The triple $\langle \mathsf{cam}, \rightarrow_{\mathsf{k}}, \sim_{\mathsf{k}}\rangle$ is a reduction machine.*

Proof. For the first clause in definition 1 notice that $s' = s''$. For the second, a routine inspection on the rules shows that $s \sim_{\mathsf{k}}\rightarrow_{\mathsf{k}} s'$ implies $s \rightarrow_{\mathsf{k}}^{*} s'$. □

All reduction steps but Call are administrative: looking values for variables in the environment (Var), and descending on a term, looking for the next redex (App, Exch). The following result says that, on each run of the CAM machine, the number of consecutive administrative steps is finite. Such a result is used in the CAM to Process encoding (section 4.3) to establish the second clause in the definition of operational correspondence (2.2).

Lemma 10. *The* CAM *machine without the* Call *rule terminates on every input.*

States: dump

$$\begin{array}{rrcl}
\mathsf{stack} & s & ::= & vc : s \mid \mathsf{nil} \\
\mathsf{controlstring} & C & ::= & M : C \mid \mathsf{ap} : C \mid \mathsf{nil} \\
\mathsf{dump} & D & ::= & \langle s, e, C, D\rangle \mid \mathsf{nil}
\end{array}$$

Reduction: $\to_{\mathsf{s}}$

$$\begin{array}{rcll}
\langle s, e, x\colon C, D\rangle & \to_{\mathsf{s}} & \langle e(x)\colon s, e, C, D\rangle & (\textsc{Var}) \\
\langle s, e, \lambda xM\colon C, D\rangle & \to_{\mathsf{s}} & \langle (\lambda xM)[e]\colon s, e, C, D\rangle & (\textsc{Abs}) \\
\langle s, e, MN\colon C, D\rangle & \to_{\mathsf{s}} & \langle s, e, M\colon N\colon \mathsf{ap}\colon C, D\rangle & (\textsc{App}) \\
\langle vc\colon (\lambda xM)[e']\colon s, e, \mathsf{ap}\colon C, D\rangle & \to_{\mathsf{s}} & \langle \mathsf{nil}, e'\{x := vc\}, M, \langle s, e, C, D\rangle\rangle & \\
 & & & (\textsc{Call}) \\
\langle vc\colon \_, \_, \mathsf{nil}, \langle s, e, C, D\rangle\rangle & \to_{\mathsf{s}} & \langle vc\colon s, e, C, D\rangle, & (\textsc{Ret})
\end{array}$$

Equivalence: $\sim_{\mathsf{s}}$

$$\langle vc, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle \sim_{\mathsf{s}} \langle \mathsf{nil}, \emptyset, \mathsf{real}(vc), \mathsf{nil}\rangle$$

Figure 3: The SECD reduction machine

Proof. Let k be the state $\langle M_0[e_0], \mathsf{rl}_1 : M_1[e_1] : \ldots \mathsf{rl}_k : M_k[e_k] : \mathsf{nil}\rangle$, where rl denotes an r or an l. Define the size of k as follows.

$$\begin{array}{rcl}
\mathrm{size}(k) & \overset{\mathsf{def}}{=} & \mathrm{size}\{M_0, \ldots, M_k\} + \mathrm{size}\{e_0, \ldots, e_k\} + \sum_{1 \le i \le k} \mathrm{size}(\mathsf{rl}_i) \\
\mathrm{size}(A) & \overset{\mathsf{def}}{=} & \sum_{a \in A} \mathrm{size}(a) \\
\mathrm{size}(\mathsf{l}) & \overset{\mathsf{def}}{=} & 1 \\
\mathrm{size}(\mathsf{r}) & \overset{\mathsf{def}}{=} & 2 \\
\mathrm{size}(MN) & \overset{\mathsf{def}}{=} & 3 + \mathrm{size}(M) + \mathrm{size}(N) \\
\mathrm{size}(\lambda xM) & \overset{\mathsf{def}}{=} & 1 + \mathrm{size}(M) \\
\mathrm{size}(x) & \overset{\mathsf{def}}{=} & 1 \\
\mathrm{size}(e) & \overset{\mathsf{def}}{=} & \sum_{x := M[e'] \in e} \mathrm{size}(M) + \mathrm{size}(e')
\end{array}$$

It is easy to see that the size of a state decreases for each application of rules Var, App, and Exch. $\square$

3.3 The SECD machine

This chapter presents the SECD machine as studied by Plotkin [11]. Machine states, or dumps, are quadruples $\langle s, e, C, D\rangle$ composed of a stack of closures, an environment, a control string, and a dump. The sets of closures and environments are those of the CAM machine (figure 2).

The *stack* is used in the SECD machine to hold value closures only: evaluated functions and arguments. The rôle of the stack in the CAM machine is played here by the control string: it holds functions and arguments waiting for evaluation.

Values at the head of the control string get transferred to the stack. Since the stack contains only value closures, rule Var looks in the environment the value closure associated to the variable at the head of the stack, and push it into the stack. On the other hand, rule Abs forms a value closure with the abstraction at the top of the stack and the current environment, and pushes the closure into the stack. Rule App replaces an application MN at the head of the control string by M, N, and ap, meaning that M and N should be evaluated in that order, and then rule Call should be used. The ap mark at the head of the control string triggers rule Call. At the top of the stack is the argument (since it was evaluated last), and, just beneath, the function. The machine gets ready for the evaluation of a new term (the body of the function) with an empty stack (meaning no subterms evaluated so far), an updated environment, the body of the function, and for dump, "what remains to be done".

To extract the term contained in a closure, we use the real function [11].

$$\mathsf{real}(M[e]) \overset{\mathsf{def}}{=} M\{\mathsf{real}(e(x_1))/x_1\}\dots\{\mathsf{real}(e(x_n))/x_n\}$$

where $\mathrm{fv}(M) = \{x_1, \dots, x_n\}$.

The reduction function over dump, written $\to_{\mathsf{s}}$, is the smallest relation satisfying the rules in figure 3. With respect to Plotkin's formulation we have changed rules App and Call, so that the function gets evaluated prior to the argument, as required by the reduction we have chosen for the call-by-value λ-calculus (section 3.1). The equivalence relation over dump, written $\sim_{\mathsf{s}}$, is the smallest equivalence relation containing the equality in figure 3.

Given a closed term M, the machine starts with the dump $\langle \mathsf{nil}, \emptyset, M, \mathsf{nil}\rangle$. It is easy to check that terminal states are of the form $\langle vc, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle$.

Example 11. *Recall the term $(\lambda x \mathbf{I} x)V$ from example 7. We have:*

$$\begin{aligned}
\langle \mathsf{nil}, \emptyset, (\lambda x \mathbf{I} x)V, \mathsf{nil}\rangle &\to_{\mathsf{s}}^{3} && (\textsc{App}, \textsc{Abs}, \textsc{Abs})\\
\langle V[\emptyset] : \lambda x \mathbf{I} x[\emptyset], \emptyset, \mathsf{ap}, \mathsf{nil}\rangle &\to_{\mathsf{s}}^{4} && \\
& && (\textsc{Call}, \textsc{App}, \textsc{Var}, \textsc{Abs})\\
\langle V[e] : \mathbf{I}[e], e, \mathsf{ap}, \langle \mathsf{nil}, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle\rangle &\to_{\mathsf{s}}^{2} && (\textsc{Call}, \textsc{Var})\\
\langle V[e], e\{y := V[e]\}, \mathsf{nil}, \langle \mathsf{nil}, e, \mathsf{nil}, \langle \mathsf{nil}, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle\rangle\rangle &\to_{\mathsf{s}} && (\textsc{Ret})\\
\langle V[e], e, \mathsf{nil}, \langle \mathsf{nil}, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle\rangle &\to_{\mathsf{s}} && (\textsc{Ret})\\
\langle V[e], \emptyset, \mathsf{nil}, \mathsf{nil}\rangle &\sim_{\mathsf{s}} && \\
\langle \mathsf{nil}, \emptyset, V, \mathsf{nil}\rangle & &&
\end{aligned}$$

where e is the environment $\{x := V[\emptyset]\}$. Notice that although $(\lambda x \mathbf{I} x)V \to_{\mathsf{v}} \mathbf{I}V$, there is no state D, such that $D \sim_{\mathsf{s}} \langle \mathsf{nil}, \emptyset, (\mathbf{I}V)[], \mathsf{nil}\rangle$ (cf. example 8). In section 4.2 we show that there the Lambda *to* SECD *correspondence is convergent (but not operational).*

The property of being halted is not preserved by equivalence: there are equivalent dumps D and D', such that D is halted but D' reduces (take $\langle \mathbf{I}[\emptyset], \emptyset, \mathsf{nil}, \mathsf{nil}\rangle$ for D, and $\langle \mathsf{nil}, \emptyset, \mathbf{I}, \mathsf{nil}\rangle$ for D'). The rule in the equivalence relation is needed in order to show that Lambda to SECD is convergent (theorem 20). We have therefore chosen not to incorporate such a (natural) condition in the definition of reduction machine (definition 1). The condition is required on the target machine only; and the SECD machine is involved as source in its main result (theorem 29).

Proposition 12. *The triple $\langle \mathsf{dump}, \to_{\mathsf{s}}, \sim_{\mathsf{s}}\rangle$ is a reduction machine.*

Proof. For the first clause in definition 1 we have that $s' = s''$. For the second, given that dumps of the form $\langle vc, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle$ do not reduce, the interesting case is when $D = \langle \mathsf{nil}, \emptyset, \mathsf{real}(vc), \mathsf{nil}\rangle \sim_{\mathsf{s}} D'$. The only rule applicable to D' is Abs, yielding $D' \to_{\mathsf{s}} D$. Since $\sim_{\mathsf{s}}$ is an equivalence relation, we have $D \sim_{\mathsf{s}} D$. $\square$

All reduction steps but Call and Ret are administrative: looking values for variables in the environment (Var), moving values from the control string into the stack (Abs), and breaking applications in the control string (App). The following result says that, on each run of the SECD machine, the number of consecutive administrative steps is finite. Such a result is used in the SECD to Context encoding (section 4.4) to establish the second clause in the definition of operational correspondence (2.2).

Lemma 13. *The* SECD *machine without rules* Call *and* Ret *terminates on every input.*

Processes

$$\begin{array}{rl}
\mathsf{name} & p,q,r,u,v,w,x,y,z \quad (\mathsf{variable} \subset \mathsf{name}) \\
\mathsf{process} & P \ ::= \ \overline{u}\tilde{v} \ \mid \ u(\tilde{x}).P \ \mid \ P \mid Q \ \mid \ \nu x P \ \mid \ !u(\tilde{x}).P \ \mid \ \mathbf{0}
\end{array}$$

Structural congruence

$$\begin{array}{rcl}
P & \equiv & Q \quad \text{if} \quad P \equiv_\alpha Q \\
P \mid \mathbf{0} & \equiv & P,\ P \mid Q \equiv Q \mid P,\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
\nu x\, \mathbf{0} & \equiv & \mathbf{0},\ \nu xy\, P \equiv \nu yx\, P \\
P \mid \nu x\, Q & \equiv & \nu x\, (P \mid Q) \quad \text{if} \quad x \notin \mathrm{fn}(P)
\end{array}$$

Reduction

$$\begin{array}{rcll}
\overline{u}\tilde{v} \mid u(\tilde{x}).P & \to & P\{\tilde{v}/\tilde{x}\} & (\textsc{Com}) \\
\overline{u}\tilde{v} \mid !u(\tilde{x}).P & \to & !u(\tilde{x}).P \mid P\{\tilde{v}/\tilde{x}\} & (\textsc{Rep}) \\
\nu x\, P & \to & \nu x\, P' \quad \text{if} \quad P \to P' & (\textsc{Scop}) \\
P \mid Q & \to & P' \mid Q \quad \text{if} \quad P \to P' & (\textsc{Par}) \\
P' & \to & Q' \quad \text{if} \quad P' \equiv P, P \to Q, \text{ and } Q \equiv Q' & (\textsc{Equiv})
\end{array}$$

Figure 4: The asynchronous π-calculus, its syntax and reduction semantics

Proof. Let D be the dump $\langle V_1[e_1] : \cdots : V_n[e_n] : \mathsf{nil}, e_0, C, D'\rangle$. Define the size of a dump as follows.

$$\begin{array}{rcl}
\mathrm{size}(D) & \stackrel{\mathsf{def}}{=} & \sum \mathrm{size}(V_i) + \mathrm{size}(\{e_0, \ldots, e_n\}) + \mathrm{size}(C) + \mathrm{size}(D') \\
\mathrm{size}(\mathsf{nil}) & \stackrel{\mathsf{def}}{=} & 0 \\
\mathrm{size}(M : C) & \stackrel{\mathsf{def}}{=} & 1 + \mathrm{size}(M) + \mathrm{size}(C) \\
\mathrm{size}(\mathsf{ap} : C) & \stackrel{\mathsf{def}}{=} & 1 + \mathrm{size}(C) \\
\mathrm{size}(MN) & \stackrel{\mathsf{def}}{=} & 1 + \mathrm{size}(M) + \mathrm{size}(N)
\end{array}$$

The remaining cases (set, $\lambda x M$, x, and e) are as in the proof of lemma 10. It is easy to see that the size of a dump decreases for each application of rules Var, App, and Abs. □

3.4 Typed π-calculus

The π-calculus equipped with input/output types constitutes the basis of the next two machines. Typed processes allow for a stronger form of the replication theorems (theorem 15 below), necessary to prove the correspondences [9]. The syntax of its asynchronous version is defined in figure 4, where the set $\mathsf{name} \setminus \mathsf{variable}$ is countable.

Processes of the form $\overline{u}\tilde{v}$ are called *messages*; u is the target, whereas the sequence of names $\tilde{v}$ represents the contents of the message. *Receptors* are processes of the form $u(\tilde{x}).P$, where u is called the location of the receptor, $\tilde{x}$ the formal parameters, and P its body. The interaction between a message $\overline{u}\tilde{v}$ and a receptor $u(\tilde{x}).P$ is the process P where names in $\tilde{v}$ replace those in $\tilde{x}$. The *parallel composition* of P and Q is written $P \mid Q$. Processes of the form $\nu x P$ introduce a new name x local, or private, to P. A *replicated receptor* $!u(\tilde{x}).P$ behaves as a persistent receptor surviving interaction with messages. Finally, $\mathbf{0}$ represents the terminated process.

We say that a process P *occurs under a prefix* when P is in the scope of a $u(\tilde{x})$. We say that a name x occurs *free* in a process P if x is not in the scope of a νx or a $u(\tilde{y}x\tilde{z})$; x occurs *bound* otherwise. The set $\mathrm{fn}(P)$ of the *free names* in P; the result of simultaneously substituting $\tilde{v}$ for the free occurrences of $\tilde{u}$ in P, denoted by $P\{\tilde{v}/\tilde{u}\}$; and *alpha equivalence*, denoted by $\equiv_\alpha$, are all defined in the standard way.

We follow the convention that, if $P_1, \ldots, P_n$ occur in a certain mathematical context, then in these processes all bound names are chosen to be different from the free names (cf. the variable convention [2]), except when otherwise mentioned.[1]

Processes of the form $\nu x_1 \ldots \nu x_n P$ are sometimes abbreviated to $\nu x_1 \ldots x_n P$. As in the λ-calculus, where the λ binds looser than application, we take the view that ν binds looser than the parallel composition: $\nu x\ P \mid Q$ means $\nu x(P \mid Q)$. This saves us a few parenthesis when writing complex process terms.

The operational semantics of processes is given by a reduction relation. Following Milner [8], reduction exploits the *structural congruence* relation over $\mathsf{process}$, written $\equiv$, and defined as the least congruence which is closed under the rules in figure 4. The *reduction relation* over $\mathsf{process}$, written $\rightarrow$, is the smallest relation satisfying the rules in the same figure.

The processes we are interested in obey the input/output type discipline. The set of input/output types is inductively defined as follows.

$$\begin{aligned} S &::= I\langle\tilde{S}\rangle \mid t \mid \mu t.S \\ I &::= \mathsf{o} \mid \mathsf{i} \mid \mathsf{b} \end{aligned}$$

Input/output types distinguish between the capability of *reading* a name ($\mathsf{i}\langle\tilde{S}\rangle$ is the type of a name that may be used only for input, and that carries names of types $\tilde{S}$), *writing* on a name ($\mathsf{o}\langle\tilde{S}\rangle$ is the type of a name that may be used only for output), and *reading and writing* on a name ($\mathsf{b}\langle\tilde{S}\rangle$).

[1]A notable exception is the Replication Theorem 15.3.

States: $\mathsf{typedprocess}$

$$\mathsf{typedprocess} \stackrel{\mathsf{def}}{=} \{P \in \mathsf{process} \mid \Gamma \vdash P,\ \text{for some } \Gamma\}$$

Reduction: $\rightarrow_{\mathsf{d}}$

$$P \rightarrow_{\mathsf{d}} P' \stackrel{\mathsf{def}}{=} (P \rightarrow P', P \rightarrow P'' \text{ implies } P' \equiv P''),\ \text{and } P \not\downarrow_{\overline{u}}$$

Equivalence: $\simeq$

$$P \simeq Q \stackrel{\mathsf{def}}{=} C[P] \dot{\sim} C[Q] \text{ for every } C \in \mathsf{typedcontext},\ \text{where } \dot{\sim} \text{ is the largest symmetric relation such that, whenever } P \dot{\sim} Q:$$

i. $P \downarrow_{\overline{u}}$ implies $Q \downarrow_{\overline{u}}$, and

ii. $P \rightarrow_{\mathsf{d}} P'$ implies $Q \rightarrow_{\mathsf{d}} \dot{\sim}\ P'$.

Figure 5: The $\mathsf{Process}$ machine

Type environments, or *typings*, denoted by Γ, Δ, are finite maps from names to types. A *typing judgment* $\Gamma \vdash P$ asserts that process P is *well typed* under the type environment Γ, or that P *obeys* Γ. We omit the definition of the $\vdash$ relation, which can be found in book [12]. If X is the set $\{x_1, \ldots, x_n\}$, we write $X : S$ to mean the typing $x_1 : S, \ldots, x_n : S$.

3.5 The π-process machine

States of the $\mathsf{Process}$ machine (figure 5) are processes typable with the input/output type assignment system.

Process reduction is nondeterministic. In order to conform to the requirement for reduction machines (definition 1), we use *deterministic reduction*, a notion based on structural congruence and on an observation predicate. For any name u, the *observation predicate* $\downarrow_{\overline{u}}$ denotes the possibility of a process immediately performing an output communication with the external environment along u. Thus, $P \downarrow_{\overline{u}}$ holds if P has a sub-term $\overline{u}\tilde{v}$ which is not underneath a prefix or in the scope of a restriction on u. We write $P \not\downarrow_{\overline{u}}$ to mean that $P \downarrow_{\overline{u}}$ does not hold for any u. Deterministic reduction is defined in figure 5.

A *context* is a process expression where the hole replaces an occurrence of $\mathbf{0}$. Contexts are denoted by C; if C is a context and P a process, we denote by $C[P]$ the process obtained by replacing P for the hole in C. Similarly to processes, we require contexts to be typed, as defined in figure 6. The equivalence for the $\mathsf{Process}$ machine is strong barbed congruence over typed processes, defined in figure 5.

Proposition 14. *The triple* $\langle \mathsf{typedprocess}, \rightarrow_{\mathsf{d}}, \simeq \rangle$ *is a reduction machine.*

States: $\mathsf{typedcontext}$

$$\mathsf{typedcontext} \overset{\mathsf{def}}{=} \{C \mid C[P] \in \mathsf{typedprocess},\ \text{for all } P \in \mathsf{typedprocess}\}$$

Reduction: $\to_{\mathsf{d}}$

$$C \to_{\mathsf{d}} C' \overset{\mathsf{def}}{=} C[P] \to_{\mathsf{d}} C'[P] \text{ for all } P \text{ s.t. } \mathrm{fn}(P) \cap \mathsf{variable} = \emptyset$$

Equivalence: $\simeq$

$$C \simeq C' \overset{\mathsf{def}}{=} C[P] \simeq C'[P] \text{ for all } P \text{ s.t. } \mathrm{fn}(P) \cap \mathsf{variable} = \emptyset$$

Figure 6: The $\mathsf{Context}$ machine

Proof. For the first clause in definition 1, we know that $\equiv \subseteq \simeq$ ([12], exercise 2.1.10). For the second, we show that $\to_{\mathsf{d}}$ commutes with $\simeq$, by recalling that strong barbed congruence coincides with strong barbed bisimulation in the asynchronous setting ([12], theorem 5.4.10), and that the commutation is part of the definition of the latter. $\square$

In proofs, as in examples, we sometimes use a stronger equivalence relation, contained in $\simeq$. The *garbage collection* relation is the equivalence that includes structural congruence and the following equality.

$$(\nu u\ !u(\tilde{x}).P) \mid Q \sim_{\mathsf{gc}} Q.$$

At the basis of the proofs of the correspondences from CAM and SECD into the π-calculus lie the (sharpened) replication theorems. We say that u, a name free in P, is used as a trigger, if P obeys a typing Γ, and the tag of the type of u in Γ is o.

Theorem 15 (Replication theorems, [12] 10.5.1(2) and (1), 2.2.28(4)). *If u is used as a trigger in P, Q, R, then*

1. $\nu u\ !u(\tilde{x}).P \mid v(\tilde{y}).Q \simeq v(\tilde{y}).(\nu u\ !u(\tilde{x}).P \mid Q)$ *for* $u \neq v$;
2. $\nu u\ !u(\tilde{x}).P \mid !v(\tilde{y}).Q \simeq !v(\tilde{y}).(\nu u\ !u(\tilde{x}).P \mid Q)$;
3. $\nu u\ !u(\tilde{x}).P \mid Q \mid R \simeq (\nu u\ !u(\tilde{x}).P \mid Q) \mid (\nu u\ !u(\tilde{x}).P \mid R)$.

3.6 The π-context machine

The notions of reduction and equivalence for typed contexts are derived from those of typed process, by taking the view that the process in the hole must not play any rôle; their definitions are in figure 6. The symbols for reduction and equivalence are those of the process machine; in each particular case, it should be clear which definition to use.

$\mathsf{load} : \mathsf{term}^0 \to \mathsf{cam}$

$$\mathsf{load}(M) \overset{\mathsf{def}}{=} \langle M[], \mathsf{nil}\rangle$$

Figure 7: The Lambda to CAM correspondence

Proposition 16. *The triple* $\langle \mathsf{typedcontext}, \to_\mathsf{d}, \simeq\rangle$ *is a reduction machine.*

Proof. Follows from the fact that $\langle \mathsf{typedprocess}, \to_\mathsf{d}, \simeq\rangle$ is a reduction machine. □

4 Four correspondences

This section presents three operational correspondences (Lambda into CAM, CAM into Process, and SECD into Context), and one convergence correspondence (Lambda into SECD). It concludes by deriving direct translations from Lambda into Process and into Context.

4.1 The Lambda to CAM correspondence

The call-by-value lambda machine and the CAM machine are defined in figures 1 and 2. The Lambda to CAM map is defined in figure 7. Since alpha-equivalent terms are loaded into CAM-equivalent states, load is a correspondence.

Lemma 17. $M \to_\mathsf{v} N$ *implies* $\langle M[], s\rangle \to_\mathsf{k}^* \sim_\mathsf{k} \langle N[], s\rangle$.

Proof. By induction on the derivation of the lambda reduction step.
Case the last rule is β: apply reduction rules App, Exch, and Call, followed by the last rule in the equivalence.

$$\langle((\lambda x M)V)[], s\rangle \to_\mathsf{k} \langle(\lambda x M)[], \mathsf{r} : V[] : s\rangle \to_\mathsf{k} \langle V[], \mathsf{l} : (\lambda x M)[] : s\rangle \to_\mathsf{k} \langle M[x := V[]], s\rangle \sim_\mathsf{k} \langle(M\{V/x\})[], s\rangle$$

Case the last rule is ν: apply reduction rules App, Exch, followed by induction, and by the first rule in equivalence.

$$\langle(VM)[], s\rangle \to_\mathsf{k} \langle V[], \mathsf{r} : M[] : s\rangle \to_\mathsf{k} \langle M[], \mathsf{l} : V[] : s\rangle \to_\mathsf{k}^* \sim_\mathsf{k} \langle M'[], \mathsf{l} : V[] : s\rangle \sim_\mathsf{k} \langle(VM')[], s\rangle$$

Case the last rule is μ: apply reduction rule App, followed by induction; conclude with the second rule in equivalence.

$$\langle(MN)[], s\rangle \to_\mathsf{k} \langle M[], \mathsf{r} : N[] : s\rangle \to_\mathsf{k}^* \sim_\mathsf{k} \langle M'[], \mathsf{r} : N[] : s\rangle \sim_\mathsf{k} \langle(M'N)[], s\rangle$$

$\mathsf{load} : \mathsf{term}^0 \to \mathsf{dump}$

$$\mathsf{load}(M) \stackrel{\mathsf{def}}{=} \langle \mathsf{nil}, \emptyset, M, \mathsf{nil} \rangle$$

Figure 8: The Lambda to SECD correspondence

$\square$

Theorem 18. *The* load *correspondence is operational.*

Proof. We show that the three clauses of definition 2.2 hold. **a)** A corollary of lemma 17. **b)** We show the contrapositive, namely, that if M diverges, then so does $\mathsf{load}(M)$. If M diverges, then rule β is used an unbounded number of times. Analysing the first case in the proof of Lemma 17, one concludes that rule Call is applied as many times. **c)** We again show the contrapositive, namely, $M \not\to_{\mathsf{v}}$ implies $\mathsf{load}(M) \not\to_{\mathsf{k}}$. If $M \not\to_{\mathsf{v}}$, then M is a closed value, say V, and $\langle V[], \mathsf{nil} \rangle \not\to_{\mathsf{k}}$. $\square$

Theorem 5.1 allows us to conclude that load is convergent as well.

Corollary 19. *The* load *correspondence is convergent.*

4.2 The Lambda to SECD correspondence

Recall the call-by-value lambda machine and the SECD machine from figures 1 and 3. The Lambda to SECD map is defined in figure 8. Since alpha-equivalent terms are loaded into SECD-equivalent dumps, load is a correspondence.

Theorem 20 (Plotkin [11], theorem 1). *The* load *correspondence is convergent.*

The only point worth notice is that Plotkin shows that if $M \downarrow_{\mathsf{v}} M'$, then $\mathsf{load}(M) \downarrow_{\mathsf{s}} \langle vc, \emptyset, \mathsf{nil}, \mathsf{nil} \rangle$ where $\mathsf{real}(vc) = M'$. We then apply the dump equivalence to obtain $\mathsf{load}(M) \downarrow_{\mathsf{s}} \sim_{\mathsf{s}} \mathsf{load}(M')$, as required.

4.3 The CAM to Process correspondence

The CAM machine and the π-process machine are the objects of definitions 2 and 5. Based on a variant of Milner's encoding, this section presents a translation of CAM states into π-processes and proves that it constitutes an operational correspondence.

Term: $\mathsf{term} \times \mathsf{name} \to \mathsf{typedprocess}$

$$\begin{aligned}
[\![x]\!]_p &\overset{\mathsf{def}}{=} \overline{p}x \\
[\![V]\!]_p &\overset{\mathsf{def}}{=} \nu u\, u := V[\emptyset] \mid \overline{p}u \\
[\![xM]\!]_p &\overset{\mathsf{def}}{=} \nu r\, [\![M]\!]_r \mid r(v).\overline{x}vp \\
[\![VM]\!]_p &\overset{\mathsf{def}}{=} \nu u\, u := V[\emptyset] \mid \nu r\, [\![M]\!]_r \mid r(v).\overline{u}vp \\
[\![MN]\!]_p &\overset{\mathsf{def}}{=} \nu q\, [\![M]\!]_q \mid q(u).(\nu r\, [\![N]\!]_r \mid r(v).\overline{u}vp)
\end{aligned}$$

Environment entry: $\mathsf{name} \times \mathsf{valueclosure} \to \mathsf{typedprocess}$

$$u := (\lambda x M)[e] \overset{\mathsf{def}}{=} [\![e]\!][\, !u(xp).[\![M]\!]_p]$$

Environment: $\mathsf{environment} \to \mathsf{typedcontext}$

$$[\![\{x_1 := vc_1, \ldots, x_n := vc_n\}]\!] \overset{\mathsf{def}}{=} \nu x_1\, x_1 := vc_1 \mid \ldots \mid \nu x_n\, x_n := vc_n \mid []$$

Stack: $\mathsf{stack} \times \mathsf{name} \to \mathsf{typedcontext} \times \mathsf{name}$

$$\begin{aligned}
[\![\mathsf{nil}]\!]_p &\overset{\mathsf{def}}{=} ([], p) \\
[\![\mathsf{l} : vc : s]\!]_p &\overset{\mathsf{def}}{=} ([\![s]\!]^p_{p'}[\nu u\, u := vc \mid \nu r\, [] \mid r(v).\overline{u}vp'], r) \\
[\![\mathsf{r} : M[e] : s]\!]_p &\overset{\mathsf{def}}{=} ([\![s]\!]^p_{p'}[\nu q\, [] \mid q(u).(\nu r\, [\![e]\!][\![M]\!]_r \mid r(v).\overline{u}vp')], q)
\end{aligned}$$

$\mathsf{load} : \mathsf{name} \to \mathsf{cam} \to \mathsf{typedprocess}$

$$\begin{aligned}
\mathsf{load}_p\langle vc, \mathsf{r} : M[e] : s\rangle &\overset{\mathsf{def}}{=} [\![s]\!]^q_p[\nu u\, u := vc \mid \nu r\, [\![e]\!][\![M]\!]_r \mid r(v).\overline{u}vp] \\
\mathsf{load}_p\langle c, s\rangle &\overset{\mathsf{def}}{=} [\![s]\!]^q_p[\![c]\!]_q
\end{aligned}$$

where names p, p', q, r, u, v are taken freshly, and the rules in $[\![M]\!]_p$ and in load must be tried from the top.

Figure 9: The CAM to Process correspondence

The encoding is described in figure 9. An *environment entry* $u := (\lambda x M)[e]$ describes the encoding of an abstraction $\lambda x M$ with environment e located at name u; it gets at u the argument x to the function as well as the channel p where to locate the result; if the evaluation of the function converges, then this channel is written with the location of the result. Replication allows the entry to be used multiple times. Values write the name they are know by (either the variable x itself, or the location of the environment entry u representing the abstraction) on the reply-to p name that locates the value. In order to mimic an application MN, we consider three cases. When M is an application, we first evaluate the function M on some fresh reply-to name q, and wait for the result u on q. Then we do the same for the argument N, thus obtaining its value v. All that remains is to invoke u with v, requesting for the resulting value to be sent on p, the location of application MN. The cases for applications xM and $(\lambda x M)N$ are obtained from the above encoding by partial evaluation. One can easily see that, for example, $\nu q\, [\![x]\!]_q \mid q(u).(\nu r\, [\![M]\!]_r \mid r(v).\overline{u}vp)$ reduces in one deterministic step to $\nu r\, [\![M]\!]_r \mid r(v).\overline{x}vp$; so we set the latter to be the image of $[\![xM]\!]_p$, rather than the general case as in Milner's encoding.

An *environment* e is encoded by translating each of its entries (recursively for the environments in the entries); the locations of the entries (the various x_i) are then made local to a hole where we place the encoding of a term M, thus providing for the encoding of a *closure* $M[e]$.

A *stack* s is compiled, by orderly unstacking its elements. The result of encoding a stack is a pair composed of a context and a name: the hole should be filled with (the translation of) a term located at the name. If $[\![s]\!]_p = (C, q)$, we write $[\![s]\!]_p^q$ to denote the context C, and use name q wherever needed. So, for example,

$$[\![s]\!]_p^q[\![c]\!]_q \text{ means '}C[[\![c]\!]_q] \text{ where } [\![s]\!]_p = (C, q)\text{'}.$$

The idea is that *subscripts represent input* to the encoding function, whereas *superscripts represent output*. The translation of states should be easy to understand by comparing the contexts in the hole of $[\![s]\!]_p$ with the encodings for VM, and MN, respectively. Notice that the hole in a context $[\![s]\!]_p$ is under no prefix.

Finally, to compile a *state* $\langle c, s\rangle$, we compile the stack s into a context (whose hole is to be located at q), and fill the hole with the encoding of closure c compiled at q. Special care must be taken when the head of the stack is an r and the closure in the state is a value closure, for the closure represents the argument to the function that follows the r; the encoding of this case should be easy to understand by referring to the encoding for VM and rule Exch in reduction (figure 2).

We argue that the encoding yields typed processes. There are two kinds of names involved:

1. *value names* comprising variables x, y, z, as well as locations of abstractions u, v, w, all in $\mathsf{variable}$; and

2. *reply-to names* p, q, r, names that return value names.

Let us call Val the type of the value names. Analysing the message $\overline{u}vp$ in the encoding of an application, we see that Val must be of the form $\mathsf{o}\langle\mathsf{Val}, T\rangle$, for T the type of the reply-to name p. Then, concentrating on the message $\overline{p}x$ in the encoding of a variable, we easily conclude that T is the type oVal. Type Val is then $\mu X.\mathsf{o}\langle X, \mathsf{o}X\rangle$. For terms, we know that ([12], lemma 15.3.14),

$$\mathrm{fv}(M) : \mathsf{Val}, p : \mathsf{oVal} \vdash [\![M]\!]_p.$$

For states, we can equally show that $p : \mathsf{oVal} \vdash \mathsf{load}(s)_p$, since states are closed (that is, for each $M[e]$ in a state, we have that $\mathrm{fv}(M) \subseteq \mathrm{dom}(e)$, cf. figure 2).

In the rest of this section we show that load is an operational correspondence. Let $u := V$ abbreviate $u := V[\emptyset]$; we start with a basic result that helps in establishing that load is a correspondence.

Lemma 21 (Substitution Lemma). $[\![x := V]\!][\![M]\!]_p \simeq [\![M\{V/x\}]\!]_p$.

Proof. By structural induction on M.
Case M is x. We have $\nu x\, x := V \mid \overline{p}x \overset{\mathsf{def}}{=} [\![V]\!]_p \overset{\mathsf{def}}{=} [\![x\{V/x\}]\!]_p$.
Case M is $y \neq x$. We have $\nu x\, x := V \mid \overline{p}y \sim_{\mathsf{gc}} \overline{p}y \overset{\mathsf{def}}{=} [\![p]\!]_y \overset{\mathsf{def}}{=} [\![y\{V/y\}]\!]_p$.
Case M is λyN. Using Replication Theorem 15.2, and induction, we have

$$\nu x\, x := V \mid \nu u\, u := \lambda yN \mid \overline{p}u \simeq \nu u\ !u(yq).(\nu x\, x := V \mid [\![N]\!]_q) \mid \overline{p}u \simeq$$
$$\nu u\ !u(yq).[\![N\{V/x\}]\!]_q \mid \overline{p}u \overset{\mathsf{def}}{=} [\![(\lambda yN)\{V/x\}]\!]_p.$$

Case M is NL, $N \notin \mathsf{value}$. Using Replication Theorem 15.3, and induction, we have

$$\nu x\, x := V \mid \nu q\, [\![N]\!]_q \mid q(u).(\nu r\, [\![L]\!]_r \mid r(v).\overline{u}vp) \simeq$$
$$\nu x\, x := V \mid \nu q\, [\![N]\!]_q \mid q(u).(\nu ry\, y := V \mid [\![L\{y/x\}]\!]_r \mid r(v).\overline{u}vp) \simeq$$
$$\nu\nu\ q[\![N\{V/x\}]\!]_q \mid q(u).(\nu r\, [\![L\{V/x\}]\!]_r \mid r(v).\overline{u}vp) \overset{\mathsf{def}}{=}$$
$$[\![N\{V/x\}L\{V/x\}]\!]_p.$$

Case M is xL, or M is $(\lambda yN)L$: similar. □

Proposition 22. *The* load_p *map is a correspondence.*

Proof. For the first two equations in the definition of $\sim_{\mathsf{k}}$, one can easily see that the encoding of the left-hand side is structural congruent to that of the right-hand side. For the last equation, use the Substitution Lemma 21. □

The following result is used to set that load is operational.

Theorem 23. $k \to_{\mathsf{k}} k'$ *implies* $\mathsf{load}_p(k) \to_{\mathsf{d}}^{*}\simeq \mathsf{load}_p(k')$, *for all* p.

Proof. A case analysis on the reduction rules. The relation that holds between $\mathsf{load}_p(k)$ and $\mathsf{load}_p(k')$ is summarised in the table below.

CAM reduction rule	Relation on processes
VAR	$\sim_{\mathsf{gc}}$
APP	$\simeq$
EXCH	$=$
CALL	$\to_{\mathsf{d}}^{2}\sim_{\mathsf{gc}}$

Case VAR: directly from the definition, collecting as garbage those entries in $[\![e]\!]$ but not in $[\![e(x)]\!]$.
Case APP: When M is an application, we have

$$\begin{aligned}
\mathsf{load}_p\langle (MN)[e], s\rangle &\overset{\mathsf{def}}{=} [\![s]\!]_p^q[\![e]\!][\nu q\,[\![M]\!]_q \mid q(u).(\nu r\,[\![N]\!]_r \mid r(v).\overline{u}vp)] \\
\mathsf{load}_p\langle M[e], \mathsf{r} : N[e] : s\rangle &\overset{\mathsf{def}}{=} [\![s]\!]_p^q[\nu q\,[\![e]\!][\![M]\!]_q \mid q(u).(\nu r\,[\![e]\!][\![N]\!]_r \mid r(v).\overline{u}vp)]
\end{aligned}$$

Start from the right-hand side of the second equation. Use replication theorem 15.1 as many times as there are entries in e, to bring the second $[\![e]\!]$ outside the scope of $q(u)$. Use then replication theorem 15.3 the same number of times to "merge" the two environments, thus obtaining the right-hand side of the first equation. Proceed similarly when M is a variable or an abstraction.
Case EXCH: directly from the definition.
Case CALL:

$$\begin{aligned}
&\mathsf{load}_p\langle V[e'], \mathsf{l} : (\lambda xM)[e] : s\rangle \overset{\mathsf{def}}{=} \\
&[\![s]\!]_{p'}^p[\nu u\,[\![e]\!][u := \lambda xM] \mid \nu r\,\underline{[\![V[e']]\!]_r} \mid \underline{r(v).\overline{u}vp'}] \to_{\mathsf{d}} \\
&[\![s]\!]_{p'}^p[\nu u\,[\![e]\!][\underline{u := \lambda xM}] \mid \nu x\,[\![e']\!][x := V] \mid \underline{\overline{u}xp'}] \to_{\mathsf{d}} \\
&[\![s]\!]_{p'}^p[\nu u\,[\![e]\!][u := \lambda xM] \mid \nu x\,[\![e']\!][x := V] \mid [\![M]\!]_{p'}] \sim_{\mathsf{gc}} \\
&\mathsf{load}_p\langle M[e\{x := V[e']\}], s\rangle
\end{aligned}$$

□

Theorem 24. *The* load_p *correspondence is operational.*

Proof. We show that the three clauses of definition 2.2 hold. **a)** Theorem 23. **b)** We show its contrapositive, namely that, if k diverges then so does $\mathsf{load}_p(k)$. Inspecting the table in the proof of theorem 23, we conclude that it suffices to show that if k diverges, then rule Call is applied an unbounded number of times. This must true since the CAM machine without rule Call converges (lemma 10). **c)** We again show the contrapositive, namely that $k \not\rightarrow_{\mathsf{k}}$ implies $\mathsf{load}(k) \not\rightarrow_{\mathsf{d}}$. Terminal states are of the form $\langle vc, \mathsf{nil}\rangle$. Clearly, $\mathsf{load}_p\langle vc, \mathsf{nil}\rangle \not\rightarrow_{\mathsf{d}}$. $\square$

Theorem 5.1 allows us to conclude that load is convergent as well.

Corollary 25. *The* load_p *correspondence is convergent.*

4.4 The **SECD** to **Context** correspondence

Recall the SECD machine and the π-context machine from figures 3 and 6. This section presents an encoding of SECD states (that is, dumps) into π-contexts and prove that it constitutes an operational correspondence.

The encoding is described in figure 10 and is based on that in reference [13]. The encoding of a *term* M is a pair composed of a context and a name: the name represents the location of the encoding of M, as seen by whatever process we place in the hole. So we see that $[\![M]\!]$ chooses the name where to locate M; whereas, in the encoding of the previous section, the name where to locate M is an argument to the encoding function. Another difference is that the result of the encoding is a context (together with a name), whereas that of Milner is a process. In contrast with the encoding of the previous section, values do not write their locations in some reply-to name. Instead they "let" the hole directly know its location: the location of a variable x is simply x; that of an abstraction is a new name u in whose scope we place both the hole and the corresponding environment entry. Applications MN first evaluate M to v, then N to w, and then invoke the function at v with the argument w and request to have the result sent to a newly created name r. Finally, they wait for the result at r, and instantiate it in the hole as u, since the process in the hole expects to see the value of MN located at u.

Similarly to the encoding of stacks in figure 9, $[\![M]\!]^v$ denotes the first component of the pair (C, v) obtained by running function $[\![\cdot]\!]$ on M, so that, in a more conventional, albeit less concise, form, we could have written:

$$[\![MN]\!] \overset{\mathsf{def}}{=} \nu r\, C[C'[\overline{v}wr]] \mid r(u).[] \text{ where } [\![M]\!] \overset{\mathsf{def}}{=} (C, v) \text{ and } [\![N]\!] \overset{\mathsf{def}}{=} (C', w).$$

Term: $\mathsf{term} \to \mathsf{typedcontext} \times \mathsf{name}$

$$\llbracket x \rrbracket \overset{\mathsf{def}}{=} ([], x)$$

$$\llbracket \lambda x M \rrbracket \overset{\mathsf{def}}{=} (\nu u\, u := \lambda x M \mid [], u)$$

$$\llbracket MN \rrbracket \overset{\mathsf{def}}{=} (\nu r\, \llbracket M \rrbracket^v \llbracket N \rrbracket^w \overline{v} w r \mid r(u).[], u)$$

Environment entry: $\mathsf{name} \times \mathsf{valueclosure} \to \mathsf{typedprocess}$

$$u := (\lambda x M)[e] \overset{\mathsf{def}}{=} \llbracket e \rrbracket[\, !u(xr).\llbracket M \rrbracket^v \overline{r} v]$$

Environment: $\mathsf{environment} \to \mathsf{typedcontext}$

$$\llbracket \{x_1 := vc_1, \ldots, x_n := vc_n\} \rrbracket \overset{\mathsf{def}}{=} \nu x_1\, x_1 := vc_1 \mid \ldots \mid \nu x_n\, x_n := vc_n \mid []$$

Stack: $\mathsf{stack} \to \mathsf{typedcontext} \times \mathsf{name}^+$

$$\llbracket vc_1 : \cdots : vc_n \rrbracket \overset{\mathsf{def}}{=} (\llbracket \{x_1 := vc_1, \ldots, x_n := vc_n\} \rrbracket, x_1 \ldots x_n)$$

load: $\mathsf{name}^+ \times \mathsf{dump} \to \mathsf{typedcontext} \times \mathsf{name}$

$$\mathsf{load}_{\vec{u}}\langle vc : \_, \_, \mathsf{nil}, \langle s, e, C, D\rangle\rangle \overset{\mathsf{def}}{=} (\llbracket s : vc \rrbracket^{\vec{v}}[\mathsf{load}^w_{\vec{u}\vec{v}}\langle \mathsf{nil}, e, C, D\rangle], w) \qquad (\textsc{Ret})$$

$$\mathsf{load}_{\vec{u}}\langle s, e, C, D\rangle \overset{\mathsf{def}}{=} (\llbracket s \rrbracket^{\vec{v}}[\mathsf{load}^w_{\vec{u}\vec{v}}\langle \mathsf{nil}, e, C, D\rangle], w) \qquad (\textsc{Stk})$$

$$\mathsf{load}_{\vec{u}}\langle \mathsf{nil}, e', M, \langle s, e, C, D\rangle\rangle \overset{\mathsf{def}}{=} (\nu r\, \llbracket e' \rrbracket \llbracket M \rrbracket^{w'} \overline{r} w' \mid r(w).\llbracket s \rrbracket^{\vec{v}} \mathsf{load}^w_{\vec{u}\vec{v}w}\langle \mathsf{nil}, e, C, D\rangle, w') \qquad (\textsc{Term})$$

$$\mathsf{load}_{\vec{u}}\langle \mathsf{nil}, e, M : C, D\rangle \overset{\mathsf{def}}{=} (\llbracket e \rrbracket \llbracket M \rrbracket^v [\mathsf{load}^w_{\vec{u}v}\langle \mathsf{nil}, e, C, D\rangle], w) \qquad (\textsc{Ctr})$$

$$\mathsf{load}_{\vec{u}vw}\langle \mathsf{nil}, e, \mathsf{ap} : C, D\rangle \overset{\mathsf{def}}{=} (\nu r\, \overline{w} v r \mid r(u').\mathsf{load}^{v'}_{\vec{u}u'}\langle \mathsf{nil}, e, C, D\rangle, v') \qquad (\textsc{Call})$$

$$\mathsf{load}_{\vec{u}v}(D) \overset{\mathsf{def}}{=} ([], v) \qquad (\textsc{Done})$$

where names $r, x_1, \ldots, x_n, u, u', v, w, w'$ are taken freshly, and the rules in load must be tried from the top.

Figure 10: The SECD to $\mathsf{Context}$ correspondence

When v is not important we write $[\![M]\!]$ to denote the first component (the context) of the pair.

Environments, *environment entries*, and *closures* are compiled in exactly the same way as their CAM counterparts, except for the new definition of the encoding of terms. *Stacks* are however compiled quite differently. Essentially a stack is encoded into the parallel composition of (the encodings of) its elements: the order is kept by the list of names returned by the encoding function. Notice that, in $[\![s]\!] = (C, \vec{u}v)$, name v denotes the (location of the) top of the stack. Further notice that name^+ in the signature of the mapping for stacks, makes load rule $\textsc{Stk}$ applicable only for non-empty stacks (thus avoiding an infinite recursion).

load proceeds by orderly compiling, first the λ-terms in the stack, and then those in the control string (rules $\textsc{Stk}$, and $\textsc{Ctr}$). To keep hold of the location of these terms, our encoding works with a non-empty sequence of names (name^+ in the signature of the encoding), rather than a single name. In this way, when time comes to compile a dump with an ap mark at the head of the control string, we know which function to call with which argument (rule $\textsc{Call}$). The $\textsc{Ret}$ reduction rule in figure 3 ignores the whole stack in the dump, but its top: before trying the general rule to compile the stack ($\textsc{Stk}$), we try the $\textsc{Ret}$ that orderly compiles the top of the stack vc as well as the stack s in the dump part of the state.

Using rules $\textsc{Ctr}$ and $\textsc{Done}$, we can easily see that the dump $\langle \mathsf{nil}, \emptyset, M, \mathsf{nil}\rangle$ (obtained by loading a closed term M into the SECD machine, cf. figure 8) is compiled into $[\![M]\!]$, as expected.

Recall that $[\![M]\!]$ yields a pair composed of a context and a name; below we write $[\![M]\!]$ to denote the context alone, and abbreviate $[\![e]\!][\![M]\!]$ to $[\![M[e]]\!]$. The rest of this section shows that load is an operational correspondence. We start with a result that helps in establishing that load is a correspondence.

Lemma 26 (Substitution Lemma). $[\![M[x := V]]\!] \simeq [\![M\{V/x\}]\!]$.

Proof. By induction on the structure of M, using the replication theorems.
Case M is the variable x; we have $[\![x[x := V]]\!] \overset{\mathsf{def}}{=} [\![V]\!]$.
Case M is a variable $y \neq x$; we have $[\![y[x := V]]\!] \overset{\mathsf{def}}{=} \nu x\, x := V \mid [] \simeq [] \overset{\mathsf{def}}{=} [\![y]\!]$, noticing that the two contexts are barbed congruent, for x, being a variable, cannot be free in the hole (cf. figure 6).
Case M is an abstraction λyN. Using the Replication theorem 15.2 (notice x being a variable is not free on the hole), and induction, we have

$$\nu x\, x := V \mid \nu v\, !v(yr).[\![N]\!]^u \overline{r}u \mid [] \simeq \nu v\, !v(yr).[\![L[x := V]]\!]^u \overline{r}v \mid [] \simeq$$
$$\nu v\, !v(yr).[\![L\{V/x\}]\!]^u \overline{r}v \mid [] \overset{\mathsf{def}}{=} [\![(\lambda yN)\{V/x\}]\!]$$

Case M is an application NL. Using the Replication Theorem 15.3, and induction twice, we have:

$$\begin{aligned}
\nu x\, x := V \mid \nu r\, [\![N]\!][\![L]\!]^w \overline{v}wr \mid r(u).[]) &\simeq \\
[\![N[x := V]]\!][\![L[x := V]]\!]^w \overline{v}wr \mid r(u).[] &\simeq \\
[\![N\{V/x\}]\!][\![L\{V/x\}]\!]^w \overline{v}wr \mid r(u).[] &\overset{\mathsf{def}}{=} \\
[\![(NL)\{V/x\}]\!]&
\end{aligned}$$

□

Proposition 27. *The* load_p *map is a correspondence, for all* p.

Proof. For the first equation in the definition of $\sim_{\mathsf{s}}$, it suffices to show that

$$[\![vc]\!] \simeq [\![\mathsf{real}(vc)]\!].$$

Let $vc = V[\{x_1 := vc_1, \ldots, x_n := vc_n\}]$, and $\mathrm{fv}(V) = \{x_1, \ldots, x_k\}$, for some $k \leq n$. We proceed by induction on the definition of real.

$$\begin{aligned}
[\![\mathsf{real}(vc)]\!] &\overset{\mathsf{def}}{=} \\
[\![V\{\mathsf{real}(vc_1)/x_1\}\ldots\{\mathsf{real}(vc_k)/x_k\}]\!] &\simeq && \text{(Substitution lemma 26)} \\
[\![V\{x_1 := \mathsf{real}(vc_1), \ldots, x_k := \mathsf{real}(vc_k)\}]\!] &\sim_{\mathsf{gc}} && \text{(Garbage collection)} \\
[\![V\{x_1 := \mathsf{real}(vc_1), \ldots, x_n := \mathsf{real}(vc_n)\}]\!] &\simeq && \text{(Induction)} \\
[\![V\{x_1 := vc_1, \ldots, x_n := vc_n\}]\!] &\overset{\mathsf{def}}{=} \\
[\![vc]\!]&
\end{aligned}$$

For the second equation, we can easily check that $[\![vc]\!] \equiv_\alpha [\![vc']\!]$, when $\mathsf{real}(vc) \equiv_\alpha \mathsf{real}(vc')$. □

The following result is used to set that load is operational.

Theorem 28. *$D \rightarrow_{\mathsf{s}} D'$ implies* $\mathsf{load}_u(D) \rightarrow_{\mathsf{d}}^{*}\simeq \mathsf{load}_u(D')$.

Proof. A case analysis on the reduction rules. The relation that holds between $\mathsf{load}_u(D)$ and $\mathsf{load}_u(D')$ is summarised in the table below.

SECD reduction rule	Relation on contexts
Var	$\sim_{\mathsf{gc}}$
Abs	$\equiv$
App	$\simeq$
Call	$\rightarrow_{\mathsf{d}}\sim_{\mathsf{gc}}$
Ret	$=$

Case Var: from $\mathsf{load}(D)$ to $\mathsf{load}(D')$, use Stk, Ctr, $\sim_{\mathsf{gc}}$, and Stk. Notice that $[\![e(x)]\!] \sim_{\mathsf{gc}} [\![x[e]]\!]$, where we have garbage collected all entries in e different from x.
Case Abs: use Stk, Ctr, $\equiv$, and Stk, in this order.
Case App: use Stk, Ctr, $\simeq$, Call, Ctr, Ctr, and Stk. For the $\simeq$ step we show that $[\![e]\!][\![M]\!]^u[\![e]\!][\![N]\!]^v \simeq [\![e]\!][\![N]\!]^u[\![M]\!]^v$. Applying the replication theorem 15.3 as many times as there are entries in e, we have:

$$[\![e]\!][\![M]\!]^u[\![e]\!][\![N]\!]^v \equiv_\alpha [\![e]\!][\![M]\!]^u[\![e\sigma]\!][\![M\sigma]\!]^v \equiv [\![e]\!][\![e\sigma]\!][\![M]\!]^u[\![M\sigma]\!]^v \simeq \\ [\![e]\!][\![M]\!]^u[\![M]\!]^v$$

where σ is an injective substitution with the domain of e, and whose codomain is fresh.
Case Call: use Stk, Call, $\rightarrow_{\mathsf{d}}$, $\sim_{\mathsf{gc}}$, $\equiv$and Term.

$$\begin{aligned}
\mathsf{load}_{\vec{u}}\langle(\lambda xM)[e'] : vc : s, e, \mathsf{ap} : C, D\rangle &\stackrel{\mathsf{def}}{=} \\
[\![s]\!]^{\vec{v}}[\![vc]\!]^{w_1}\underline{[\![(\lambda xM)[e']]\!]^{w_2}[\nu r\ \overline{w_2}w_1r} \mid r(w).\mathsf{load}^{v'}_{\vec{u}\vec{v}w}\langle \mathsf{nil}, e, C, D\rangle] &\rightarrow_{\mathsf{d}}\sim_{\mathsf{gc}} \\
[\![s]\!]^{\vec{v}}[\![vc]\!]^{w_1}[\nu r[\![M\{w_1/x\}[e']]\!]^{w'}\overline{r}w' \mid r(w).\mathsf{load}^{v'}_{\vec{u}\vec{v}w}\langle \mathsf{nil}, e, C, D\rangle]) &\equiv \\
[\![s]\!]^{\vec{v}}[\nu r[\![M[e'\{x := vc\}]]\!]^{w'}\overline{r}w' \mid r(w).\mathsf{load}^{v'}_{\vec{u}\vec{v}w}\langle \mathsf{nil}, e, C, D\rangle] &\stackrel{\mathsf{def}}{=} \\
\mathsf{load}_{\vec{u}}\langle \mathsf{nil}, e'\{x := vc\}, M, \langle s, e, C, D\rangle\rangle &
\end{aligned}$$

For the $\simeq$ step we collect the w_2 environment entry. For the $\equiv$ step, it follows from the encoding that the context $[\![vc]\!]^{w_1}[\![M\{w_1/x\}[e']]\!]^{w'}$ is structural congruent to the context $[\![M[e'\{x := vc\}]]\!]^{w'}$.
Case Ret: use Ret followed by Stk. □

Theorem 29. *The* load_u *correspondence is operational, for all* u.

Proof. We show that the three clauses of definition 2.2 hold. **a)** Theorem 28. **b)** We show its contrapositive, namely, if D diverges then so does $\mathsf{load}_u(D)$. Inspecting the table in the proof of theorem 28, we conclude that it suffices to show that if M diverges, then rule Call is applied an unbounded number of times. This must true since the SECD machine without the Call converges (lemma 13). **c)** We again show the contrapositive, namely that $D \not\rightarrow_{\mathsf{s}}$ implies $\mathsf{load}_u D \not\rightarrow$, by analyzing the dumps that do not reduce. They are of the form $\langle vc, \_, \mathsf{ap} \colon \_, \_\rangle$, $\langle \mathsf{nil}, \_, \mathsf{ap} \colon \_, \_\rangle$, $\langle \mathsf{nil}, \_, \mathsf{nil}, \_\rangle$, $\langle \_, \_, \mathsf{nil}, \mathsf{nil}\rangle$, and nil. Clearly their encodings do not reduce. □

Theorem 5.1 allows us to conclude that load is convergent as well.

Corollary 30. *The* load_u *correspondence is convergent.*

4.5 Lambda to Pi, directly

We can easily see that the Lambda to Process correspondence is operational. In fact, we know that both the Lambda to CAM and the CAM to Process correspondences are operational, and that the composition of two operational correspondences is operational (theorems 18, 24, and 5.2).

Similarly, the Lambda into Context correspondence is convergent. This follows from the following results: the Lambda to SECD correspondence is convergent, the SECD to Context is operational, the composition of a convergent with an operational correspondence is convergent (theorems 20, 29, and 5.4).

Finally, given that operational correspondences are convergent (theorem 5.1), we conclude that the Lambda to Process correspondence is convergent.

Theorem 31. *1. The* Lambda *to* Process *correspondence is operational;*

2. The Lambda *to* Context *correspondence is convergent.*

3. The Lambda *to* Process *correspondence is convergent.*

The remainder of this section studies direct encodings of the λ-calculus into the π-calculus, establishing the adequacy of the two encodings. The task is simplified since, in both cases, we proved the respective Substitution Lemmas (21, 26). The direct proofs below also allow us to quantify the number of π-steps involved in the simulation of a λ-step: exactly two in each case (on average for the Context case). We start with the Process case.

Theorem 32. *Let* $[\![\cdot]\!]$ *be the map defined in figure 9.* $M \to_{\mathsf{v}} N$ *implies* $[\![M]\!] \to_{\mathsf{d}}^2 \simeq [\![N]\!]$.

Proof. By transition induction.
Case the last rule is β and M is $(\lambda x M)y$.

$$[\![(\lambda x M)y]\!]_p \to_{\mathsf{d}}^2 (\nu u\, u := \lambda x M) \mid [\![M\{y/x\}]\!] \sim_{\mathsf{gc}} [\![M\{y/x\}]\!].$$

Case the last rule is β and M is $(\lambda x M)V$, for V an abstraction. Using the Substitution Lemma 21, we have:

$$[\![(\lambda x M)V]\!]_p \to_{\mathsf{d}}^2 (\nu u u := \lambda x M) \mid \nu v v := V \mid [\![M\{v/x\}]\!]_p \sim_{\mathsf{gc}}\simeq [\![M\{V/x\}]\!]_p$$

Case the last rule is ν. Directly by induction, both for V a variable and an abstraction.
Case the last rule is μ. Directly by induction. □

The above result should be contrasted with the one obtained by Sangiorgi and Walker for the unoptimized version of the translation from terms to processes. When one omits the particular cases of the translation of applications ($[\![xM]\!]_p$ and $[\![VM]\!]_p$ in figure 9) all we can say about processes $[\![M]\!]_p$ and $[\![N]\!]_p$ is that $[\![M]\!]_p \rightarrow_{\mathsf{d}}^{n+3}\simeq P$, and $[\![N]\!]_p \rightarrow_{\mathsf{d}}^{n} P$, for some $n \geq 0$ and some P ([12], lemma 15.3.22). Number n represents the nesting depth of the redex in M, that is, the number of applications one must cross to reach the redex, when descending the derivation tree of M. The optimized encoding obviates the π-steps necessary to reach the π-redex corresponding to the λ-redex.

Corollary 33 (Adequacy of the process encoding). *Let $[\![\cdot]\!]$ be the map defined in figure 9. $M \downarrow_{\mathsf{v}}^{n} N$ implies $[\![M]\!] \downarrow_{\mathsf{d}}^{2n}\simeq [\![N]\!]$.*

Proof. From theorem 32, and the fact that $\simeq$ commutes with $\rightarrow_{\mathsf{d}}$ (proof of proposition 14). □

Now for the Context machine.

Theorem 34 (Adequacy of the context encoding). *Let $[\![\cdot]\!]$ be the map defined in figure 10. $M \downarrow_{\mathsf{v}}^{n} N$ implies $[\![M]\!] \downarrow_{\mathsf{d}}^{2n}\simeq [\![N]\!]$.*

Proof. By induction on n. When n is zero, $M = N$, and we are done. When n is positive, M is an application, say M_1M_2. Let $M_1 \downarrow_{\mathsf{v}}^{n_1} \lambda xL$, $M_2 \downarrow_{\mathsf{v}}^{n_2} V$, and $L\{V/x\} \downarrow_{\mathsf{v}}^{n_3} N$, with $n = n_1 + n_2 + n_3 + 1$. We have $[\![M_1M_2]\!] \stackrel{\mathsf{def}}{=} (\nu r\, [\![M_1]\!]^v[\![M_2]\!]^w\overline{v}wr \mid r(u).[], u)$, and

$$\begin{aligned}
\nu r\, [\![M_1]\!]^v[\![M_2]\!]^w\overline{v}wr \mid r(u).[] &\rightarrow_{\mathsf{d}}^{2(n_1+n_2)}\simeq && \text{(Induction)}\\
\nu r\, \nu v\, \underline{v := \lambda xL} \mid \nu w\, w := V \mid \underline{\overline{v}wr} \mid r(u).[] &\rightarrow_{\mathsf{d}}\sim_{\mathsf{gc}} \\
\nu r\, \nu w\, w := V \mid [\![L\{c/x\}]\!]^{w'}\overline{r}w' \mid r(u).[] &\simeq && \text{(Subst. Lemma 21)}\\
\nu r\, [\![L\{V/x\}]\!]^{w'}\overline{r}w' \mid r(u).[] &\rightarrow_{\mathsf{d}}^{2n_3}\simeq && \text{(Induction)}\\
\nu r\, [\![N]\!]^{w'}\overline{r}w' \mid r(u).[] &\stackrel{\mathsf{def}}{=} \\
\nu r\, \nu w\, 'w' := N \mid \underline{\overline{r}w'} \mid \underline{r(u).[]} &\rightarrow_{\mathsf{d}}\equiv \\
\nu u\, u := N \mid []
\end{aligned}$$

Finally $(\nu u\, u := N \mid [], u) \stackrel{\mathsf{def}}{=} [\![N]\!]$. □

4.6 Lambda to Pi, soundness

It is not difficult to show that both interpretations of the λ-calculus into the π-calculus are sound, by following Sangiorgi and Walker [12].

For the equivalence in λ-terms (equipped with the call-by-value strategy), we pick the *Morris context equivalence*, also called *observation equivalence.*

Definition 35. *Two closed λ-terms M, N are* observationally equivalent *if $C[M] \downarrow_{\mathsf{v}}$ iff $C[N] \downarrow_{\mathsf{v}}$, for each closed λ-context C. In this case we write $M \simeq_{\mathsf{v}} N$.*

The equivalence for π-calculus *processes* is the weak version of barbed congruence, defined by replacing, in figure 5, reduction $\rightarrow_{\mathsf{d}}$ with the relation $\rightarrow^*_{\mathsf{d}}$, and the observation predicate $\downarrow_a$ with the predicate $\rightarrow^*_{\mathsf{d}}\downarrow_a$. We write $\cong_{\mathsf{p}}$ for the relation thus obtained. The equivalence for π-calculus *contexts* is obtained from the strong version, by replacing, in figure 6, $\simeq$ by $\cong_{\mathsf{p}}$. We use symbol $\cong_{\mathsf{c}}$ for the relation thus obtained.

Then we can easily show soundness from the adequacy results of section 4.5, by following the proof of the same result for the unoptimized version of the process-encoding ([12], theorem 17.3.3)

Theorem 36. *Let $M, N \in \mathsf{term}^0$, and $M \simeq_{\mathsf{v}} N$. Then,*

1. *$[\![M]\!] \cong_{\mathsf{p}} [\![N]\!]$, for $[\![\cdot]\!]$ the map defined in figure 9;*
2. *$[\![M]\!] \cong_{\mathsf{c}} [\![N]\!]$, for $[\![\cdot]\!]$ map defined in figure 10.*

Completeness, on the other hand is not be expected to hold. See discussion in book [12], after theorem 17.3.3.

5 Further work

The impact of the π-encodings proposed here on actual, π-based, programming languages should be analyzed [10, 15]. Also, further optimizations for particular patterns of λ-terms (e.g. recursive function definitions) could be pursued.

The encodings of the environment machines opens perspectives of encoding others machines, thus providing for the study of other λ-reduction strategies in the realm of π-calculus. In this respect, the context encoding seems to be tightly connected to the call-by-value strategy, remaining unclear how to apply it to call-by-name. It remains open whether the two λ-encodings are barbed congruent.

Sangiorgi and Walker presents Milner's encoding using a continuation-passing style [12]. It should be interesting to investigate whether there is a CPS transform that yields the encoding that goes via the SECD machine.

Acknowledgments. Special thanks to D. Sangiorgi, G. Boudol, R. Amadio, and A. Ravara for fruitful discussions. The Mimosa project at INRIA, Sophia-Antipolis, and the Wednesday Morning Club at the University of

Lisbon provided important feedback. This work is partially supported by project PRAXIS P/EEI/120598/98 DiCoMo.

References

[1] Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda-Calculi.* Number 46 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[2] H. P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1974. Revised edition.

[3] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.

[4] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[5] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of ECOOP '91*, volume 512 of *LNCS*, pages 133–147. Springer, July 1991.

[6] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4), 1964.

[7] Robin Milner. Functions as processes. Rapport de Recherche RR-1154, INRIA Sophia-Antipolis, 1990. Final version in [8].

[8] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992. Previous version as Rapport de Recherche 1154, INRIA Sophia-Antipolis, 1990, and in *Proceedings of ICALP '91*, LNCS 443.

[9] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extract appeared in *Proceedings of LICS '93*: 376–385.

[10] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing. MIT Press, May 2000.

[11] G.D. Plotkin. Call-by-name and call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

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

[13] Vasco T. Vasconcelos. Typed concurrent objects. In *8th European Conference on Object-Oriented Programming*, volume 821 of *LNCS*, pages 100–117. Springer-Verlag, July 1994.

[14] Vasco T. Vasconcelos. Processes, functions, datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999.

[15] Vasco T. Vasconcelos and Luís Lopes. The TyCO programming language—compiler and virtual machine. URL: http://www.ncc.up.pt/~lblopes/tyco, 1988–2002.