The call-by-value lambda-calculus, the SECD machine, and the pi-calculus

Vasco T. Vasconcelos

DI–FCUL TR–00–3

May 2000

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/biblioteca/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Abstract

We present an encoding of the call-by-value lambda-calculus into the pi-calculus, alternative to the well-known Milner's encodings. We show that our encoding is barbed congruent (under typed contexts) to Milner's "light" encoding, and that it takes two pi-steps to mimic a beta-reduction for normalizing terms.

We describe a translation of Plotkin's SECD machine into the pi-calculus, and show that there is an operational correspondence between a SECD machine and its encoding.

Equipped with a notion of state-based machine and two kinds of correspondences between them, we compare the encodings of the call-by-value lambda-calculus and the SECD machine into the pi-calculus.

Contents

Chapter 1

Introduction

Most concurrent programs include large pieces of sequential code. While the efficient compilation schemes for sequential (functional) languages are certainly not based on the pi-calculus, the sequential code of pi-based concurrent programming languages is often interpreted in a pi-calculus machine [6, 17]. For a smooth integration of functional computations into π-based concurrent languages, λ-terms are compiled into π-processes and run as ordinary processes. In this setting, the efficiency of the encoding, more precisely, the number of reduction steps in the π-calculus needed to mimic a β-reduction, becomes of paramount importance.

We study an encoding of the call-by-value λ-calculus [14] into the π-calculus [3, 10], alternative to the ones proposed by Milner [8]. The encoding is not new; it was presented before in the context of idioms for concurrent programming languages [18, 19]; we now analyze its behavior.

The SECD machine [14] was studied by Plotkin as a simplification of Landin's ISWIM [5] machine, and reflects the behaviour of stack-based implementations of functional languages. We present an encoding of the SECD machine in the π-calculus, based on the above mentioned encoding of the call-by-value λ-calculus into the π-calculus.

To compare the call-by-value λ-calculus, the SECD machine and the two encodings of λ into π, we set up four state-based machines, each composed of a set of states, a transition relation, and an equivalence relation. Given two such machines, and a pair of encoding/decoding mappings between them, we define two kinds of correspondences: the usual operational correspondence, and a convergence correspondence where a state and its image can only be compared when reduction is no longer possible.

In this setting, we show that our encoding of the call-by-value λ-calculus is convergent (but not operational), and that the encoding of the SECD machine is operational, thus revealing that our encoding closely behaves as stack-based implementations (with call-return) of functional languages. From the literature we know that the encoding of the call-by-value λ-calculus into SECD is convergent [14], and Milner's encoding is operational [16]. Section 2.3 summarizes the correspondences.

Our encoding is similar to Milner's "light version" [4, 9, 12, 16]: it compiles a λ-term into a π-process, given a name (the location of the term) that may be used to interacted with the compiled term. Furthermore, encoded terms only

behave correctly when placed in controlled contexts, namely contexts governed by input-output types [12]. However, contrary to the above mentioned encodings that, given a λ-term and a name, return a π-process ready to run, we further require the scope (in the form of a process) where the encoded term is to be used. In other words, our compilation scheme returns a context with a hole that defines the scope where the encoded term may be accessed. In order to run the encoded term we need to fill the hole with a process that invokes the term. In this way, not only do we exert a finer control on the scope where the encoded term is to be used, but, more importantly, we may directly instantiate in the hole the location of the term. In contrast, Milner's encoding sends this location to some user-defined receptor that sets up the scope. This mechanism ultimately allows us to save one message exchange per β-reduction, with respect to Milner's encoding.

The usage of contexts also simplifies the incorporation of pieces of sequential code in concurrent programs. The π-calculus allows names to carry names only; we write $\overline{x}y$ to denote the sending of name y on channel x. We would often like to send an arbitrary λ-term on a channel; for example, we would like to write $\overline{x}M$ to denote the sending of the (cbv normal-form of the) λ-term M on channel x. With our scheme, such an "extended" π-process can be interpreted as the process obtained by compiling M into a process located at some (fresh) name a valid only in the scope denoted by the process $\overline{x}a$, written $[\![M]\!]_a\overline{x}a$.

Specifically, we show that

1. the encoding of the call-by-value λ-calculus is convergent;
2. the encoding of the SECD machine is operational;
3. the cbv λ-encoding is barbed congruent to Milner's "light" encoding.

The outline of the paper is as follows. The next section introduces the notion of operational and convergence correspondences and presents a summary of the correspondences discussed in the paper. The following three sections briefly introduce the call-by-value λ-calculus, the SECD machine, and the π-calculus. Then, section 6 studies the encoding of the call-by-value λ-calculus; and section 7 compares to Milner's encoding. Section 8 is dedicated to the study of the SECD encoding. Section 9 concludes the paper.

Chapter 2

The correspondences

This chapter introduces the notions of machines and correspondences between them, and summarizes the correspondences discussed in the paper.

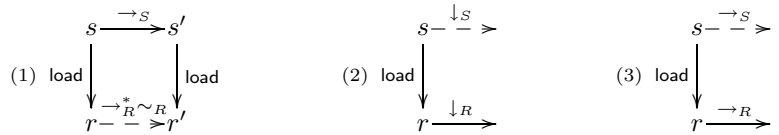
2.1 Machines and correspondences

Machines are given by a set of states, a transition function, and an equivalence relation. We write $\langle S, \rightarrow, \sim \rangle$ for the machine composed of set of states S, a function $\rightarrow$ in $S \times S$, and an equivalence relation $\sim$ in $S \times S$, such that the relation $\rightarrow^* \sim$ is transitive.[1] The equivalence relation we are thinking of is a "strong" one: alphabetic equivalence in the lambda-calculus, strong bisimulation in the π-calculus.

For a given machine $\langle S, \rightarrow, \sim \rangle$, we say that state s *converges* to state s' in n steps ($n \geq 0$), and write $s \downarrow^n s'$, if $s \rightarrow s_1 \rightarrow \ldots \rightarrow s_n = s' \not\rightarrow$. When n is not important, we write $s \downarrow s'$, and when s' is also not important, we write $s \downarrow$. Notice that $\downarrow$ is a partial function.

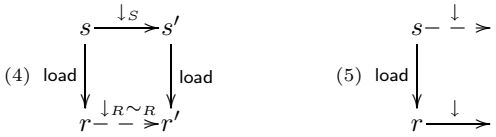
Given two machines $\langle S, \rightarrow_S, \sim_S \rangle$, $\langle R, \rightarrow_R, \sim_R \rangle$, a function $\mathsf{load} : S \rightarrow R$ is called a *correspondence*. We distinguish two kinds of correspondences. In the below diagrams solid arrows describe conditions, dashed arrows conclusions. For example, the first diagram reads: for all s, s', r, r', if $s \rightarrow_S s'$ and $\mathsf{load}(s) = r$ and $\mathsf{load}(s') = r'$, then $r \rightarrow_R^* \sim_R r'$.

1. A correspondence is *operational* if



[1] $\rightarrow^*$ is the transitive and reflexive closure of $\rightarrow$.

2. A correspondence is *convergent* if

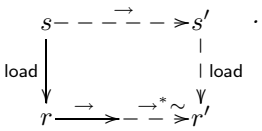


Notice that the equivalence relation of the source machine is not used in the definitions.

2.2 Some properties of convergences

The following diagram is sometimes used in the definition of operational correspondence.

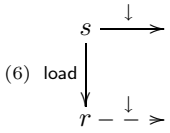
Proposition 1. *For* load *an operational correspondence,*



Proof. Consider diagrams (3) and (1). At the bottom, the two diagrams read $r \to$ and $r \to^* \sim r'$, respectively. Since $\to$ is a function, there must be a r'' such that $r \to r'' \to^* r'$. □

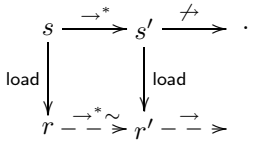
For the operational case note that, together with diagram (1), diagram (3) is stronger than the usual requirement that '$s \downarrow$ implies $\mathsf{load}(s) \downarrow$'.

Proposition 2. *For* load *an operational correspondence*



Proof. Suppose that $s \downarrow^n$. Compose n copies of diagram (1) with the contrapositive of diagram (3). The bottom line reads $r \to^* \sim r_1 \to * \sim \ldots \to^* \sim r_n \not\to$. Use the hypothesis that $\to^* \sim$ is closed under transitivity, to get $r \to^* \sim r_n \not\to$. □

Notice that (3) is weaker than (6). Using diagrams (1) and (6), we may have

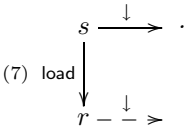


Proposition 3 (Adequacy). *For* load *an operational correspondence,* $s \downarrow$ *iff* $\mathsf{load}(s) \downarrow$.

Proof. Diagrams (2) and (6). □

Now for the convergent case.

Proposition 4. *For* load *a convergence correspondence,*

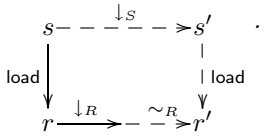


Proof. Directly from diagram (4). □

Proposition 5 (Adequacy). *For* load *a convergence correspondence,* $s \downarrow$ *iff* $\mathsf{load}(s) \downarrow$.

Proof. Diagrams (5) and (7). □

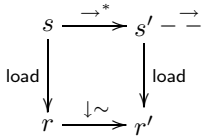
Proposition 6. *For* load *a convergence correspondence,*



.

Proof. Start with diagram (5); then use diagram (4). Remember that $\downarrow$ is a (partial) function. □

Note that the above result does not imply that



. The SECD correspondence in chapter 4 constitutes a counter-example. The main result concerning correspondences say that operational correspondences are also convergence correspondences.

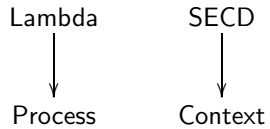
Proposition 7. *Operational correspondences are also convergence correspondences.*

Proof. Diagram (5) is diagram (2). To show that diagram (4) holds, suppose that $s \downarrow^n$. Build a diagram by placing n copies of (1) side-by-side; end with a copy of diagram (3). The bottom line reads $r \rightarrow^* \sim r_1 \rightarrow * \sim \ldots \rightarrow^* \sim r_n \not\rightarrow$; use the hypothesis that $\rightarrow_R^* \sim_R$ is closed under transitivity, to get $r \rightarrow^* \sim r_n \not\rightarrow$. □

2.3 Summary of the correspondences

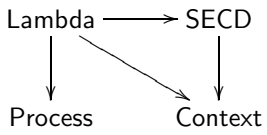
This section summarizes the correspondences between the various machines described in the paper. The four machines under consideration are the lambda-calculus equipped with the call-by-value strategy (Lambda, chapter 3), the SECD machine (SECD, chapter 4), and two machines based on the pi-calculus (Context, section 6.3, and Process, section 7.1).

For two machines M_1, M_2, write $M_1 \rightarrow M_2$ if there is a correspondence from M_1 to M_2. For the *operational* case, we have



where the correspondence Lambda → Process is the object of Theorem 25, and the correspondence SECD → Context is shown in Theorem 30.

For the *convergence* case we have



where the correspondence Lambda → SECD is the object of Theorem 1 in Plotkin [14] (cf. chapter 4). The two down arrows follow from proposition 7 and the respective arrows in the operational correspondence. The correspondence Lambda → Context is obtained indirectly via the SECD machine (that is, by composing the results for Lambda → SECD and SECD → Context), or directly by theorem 17.

Chapter 3

The call-by-value λ-calculus

This chapter introduces Plotkin's call-by-value λ-calculus [14]. We presuppose a countable set $\mathsf{variable}$ of variables, and let x, y, z range over it. The sets value and term, of values and terms, are inductively defined as follows.

$$\begin{array}{rcl} V & ::= & x \mid \lambda x M \\ M & ::= & V \mid MN \end{array}$$

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 $\mathrm{fv}(M)$ 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 $\mathrm{fv}(M) = \emptyset$. The set of closed lambda-terms is denoted by term^0.

The *reduction relation* over term, written $\to_{\mathsf{v}}$, is the smallest relation over term satisfying the following rules.

$$(\lambda x M)V \to M[V/x] \qquad (\beta)$$

$$\frac{M \to M'}{VM \to VM'} \qquad (\textsc{AppR})$$

$$\frac{M \to M'}{MN \to M'N} \qquad (\textsc{AppL})$$

The $\to_{\mathsf{v}}$ relation is Plotkin's left reduction: "If $M \to_{\mathsf{v}} N$, then N is gotten from M by reducing the leftmost redex, not in the scope of a λ" [14].

In the sequel we are interested in closed lambda-terms. Consequently, the (call-by-value) *lambda machine*, Lambda, we need is $\langle \mathsf{term}^0, \to_{\mathsf{v}}, \equiv_\alpha \rangle$. Notice that, since states are closed terms, saying that $M \downarrow^n N$ is saying that $M \to^n N$ and N is a value.

Chapter 4

The SECD machine

This chapter presents the SECD machine, a simplified formulation of the Landin's ISWIM machine [5], as studied by Plotkin [14].

The SECD machine is given by a set of states and a transition function. 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, $\mathsf{closure}$ and $\mathsf{environment}$, are defined mutually recursively. Symbol Cl is used to range over closures, and E over environments.

1. $\{\langle x_i, \mathrm{Cl}_i\rangle \mid i = 1, n\}$ is in $\mathsf{environment}$, if $x_1, \ldots, x_n$ are distinct variables and $n \geq 0$; in this case, the domain of the environment, $\mathrm{dom}(E)$, is the set $\{x_1, \ldots, x_n\}$;

2. $\langle M, E\rangle$ is in $\mathsf{closure}$, if $\mathrm{fv}(M) \subseteq \mathrm{dom}(E)$.

Notice that an empty environment constitutes the base of recursion: if M is a closed term, then $\langle M, \emptyset\rangle$ is a closure. The sets of control strings and of dumps, $\mathsf{controlstring}$ and dump, are defined as follows. Letter C is used to range over control strings and D over dumps.

1. $\mathsf{controlstring} \stackrel{\mathsf{def}}{=} (\mathsf{term} \cup \{\mathsf{ap}\})^*$, where $\mathsf{ap} \notin \mathsf{term}$;

2. nil is in dump, and so is $\langle S, E, C, D\rangle$.

Environment update is captured by the operation $E\{\mathrm{Cl}/x\}$, denoting the unique environment E' such that $E'(y) = E(y)$ if $y \neq x$, and $E'(x) = \mathrm{Cl}$ otherwise. The term represented by a closure is given by the $\mathsf{real} : \mathsf{closure} \to \mathsf{term}$ function, defined as

$$\mathsf{real}(\langle M, E\rangle) \stackrel{\mathsf{def}}{=} M[\mathsf{real}(E(x_1))/x_1] \ldots [\mathsf{real}(E(x_n))/x_n]$$

where $\mathrm{fv}(M) = \{x_1, \ldots, \mathrm{n}_n\}$.

The transition function over dump, written $\rightarrow_{\mathrm{SECD}}$, is defined by the follow-

ing rules.

$$\langle S, E, x\colon C, D\rangle \to \langle E(x)\colon S, E, C, D\rangle \qquad \text{(Var)}$$
$$\langle S, E, \lambda x M\colon C, D\rangle \to \langle \langle \lambda x M, E\rangle\colon S, E, C, D\rangle \qquad \text{(Abs)}$$
$$\langle S, E, MN\colon C, D\rangle \to \langle S, E, N\colon M\colon \mathsf{ap}\colon C, D\rangle \qquad \text{(App)}$$
$$\langle \langle \lambda x M, E'\rangle\colon \mathrm{Cl}\colon S, E, \mathsf{ap}\colon C, D\rangle \to \langle \mathsf{nil}, E'\{\mathrm{Cl}/x\}, M, \langle S, E, C, D\rangle\rangle \qquad \text{(Call)}$$
$$\langle \mathrm{Cl}\colon S, E, \mathsf{nil}, \langle S', E', C', D'\rangle\rangle \to \langle \mathrm{Cl}\colon S', E', C', D'\rangle, \qquad \text{(Ret)}$$

To translate a lambda term into a dump, use the $\mathsf{term\text{-}to\text{-}dump} : \mathsf{term} \to \mathsf{dump}$ function defined as follows:

$$\mathsf{term\text{-}to\text{-}dump}(M) \stackrel{\mathsf{def}}{=} \langle \mathsf{nil}, \emptyset, M, \mathsf{nil}\rangle$$

If a SECD machine halts at a dump of the form $\langle \mathrm{Cl}, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle$, then we read the result by identifying the term represented by the closure Cl, using function real. Thus the equivalence on dumps, $\equiv_{\mathrm{SECD}}$, is the smallest relation that includes the pairs

$$(\langle \mathsf{nil}, \emptyset, M, \mathsf{nil}\rangle, \langle \mathrm{Cl}, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle)$$

for all terms M and closures Cl such that $\mathsf{real}(\mathrm{Cl}) \equiv_\alpha M$. The *SECD machine*, SECD, is the triple $\langle \mathsf{dump}, \to_{\mathrm{SECD}}, \equiv_{\mathrm{SECD}}\rangle$. Notice that if we load the machine with an abstraction V, we get the dump $\langle \mathsf{nil}, \emptyset, V, \mathsf{nil}\rangle$. Such a dump both reduces and is equivalent to $\langle \langle V, \emptyset\rangle, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle$.

Theorem 8 ([14], Theorem 1). *The* $\mathsf{term\text{-}to\text{-}dump}$ *correspondence is convergent.*

Example 9. Consider the term $(\lambda x \mathbf{I} x)V$ where V is a closed value. We have

$$\begin{aligned}
\mathsf{term\text{-}to\text{-}dump}((\lambda x \mathbf{I} x)V) &\to^*_{\mathrm{SECD}} \\
\langle \langle \lambda x \mathbf{I} x, \emptyset\rangle : \langle V, \emptyset\rangle, \emptyset, \mathsf{ap}, \mathsf{nil}\rangle &\to^*_{\mathrm{SECD}} \\
\langle \langle \mathbf{I}, E\rangle : \langle V, E\rangle, E, \mathsf{ap}, \langle \mathsf{nil}, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle\rangle &\to^*_{\mathrm{SECD}} \\
\langle \langle V, E\rangle, E\{\langle V, E\rangle/y\}, \mathsf{nil}, \langle \mathsf{nil}, E, \mathsf{nil}, \langle \mathsf{nil}, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle\rangle\rangle &\to_{\mathrm{SECD}} \\
\langle \langle V, E\rangle, E, \mathsf{nil}, \langle \mathsf{nil}, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle\rangle &\to_{\mathrm{SECD}} \\
\langle \langle V, E\rangle, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle &
\end{aligned}$$

where E is the environment $\{\langle x, \{\langle V, \emptyset\rangle\}\rangle\}$. The second and the third line correspond to Call transitions; the fourth and fifth to Ret transitions. Then, we read the result to get $\mathsf{real}(\langle V, E\rangle) = V$ as expected.

Chapter 5

The π-calculus

This section briefly introduces the (asynchronous) π-calculus to the extent needed for this presentation, namely, the syntax, the reduction semantics, i/o-types, and barbed congruence.

5.1 Syntax

Fix a countable set name of names, and let lower-case letters range over this set. The set process of processes is inductively defined as

$$P \;::=\; \overline{a}\tilde{b} \;\mid\; a(\tilde{x}).P \;\mid\; P \mid Q \;\mid\; \nu x P \;\mid\; !a(\tilde{x}).P \;\mid\; \mathbf{0}$$

where the names in the sequence of names $\tilde{x}$ are pairwise distinct.

Processes of the form $\overline{a}\tilde{b}$ are called *messages*; a is the target, whereas the sequence of names $\tilde{b}$ represents the contents of the message. *Receptors* are processes of the form $a(\tilde{x}).P$, where a is called the location of the receptor, $\tilde{x}$ the formal parameters, and P its body. The interaction between a message $\overline{a}\tilde{b}$ and a receptor $a(\tilde{x}).P$ is the process P where names in $\tilde{b}$ 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* $!a(\tilde{x}).P$ 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 $a(\tilde{x})$.

We follow the convention that the scope of νx extends as much to the right as possible, so that, for example, $\nu x\, P \mid Q$ denotes the process $\nu x\, (P \mid Q)$.

5.2 Reduction semantics

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

We also 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 [1]).

The operational semantics of processes is given by a *reduction relation*. Following Milner [9], reduction exploits the *structural congruence* relation over process, written $\equiv$, and defined as the smallest relation containing alpha equivalence that satisfies the following rules.

1. $P \mid \mathbf{0} \equiv P$, $P \mid Q \equiv Q \mid P$, and $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$;
2. $\nu x\, \mathbf{0} \equiv \mathbf{0}$, and $\nu xy\, P \equiv \nu yx\, P$;
3. $P \mid \nu x\, Q \equiv \nu x\, P \mid Q$ if $x \notin \mathrm{fn}(P)$.

The *reduction relation* over process, written $\rightarrow$, is the smallest relation satisfying the following rules.

$$\overline{a}\tilde{b} \mid a(\tilde{x}).P \rightarrow P[\tilde{b}/\tilde{x}] \qquad \text{(Com)}$$

$$\overline{a}\tilde{b} \mid !a(\tilde{x}).P \rightarrow !a(\tilde{x}).P \mid P[\tilde{b}/\tilde{x}] \qquad \text{(Rep)}$$

$$\frac{P \rightarrow P'}{\nu x\, P \rightarrow \nu x\, P'} \qquad \text{(Scop)}$$

$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \text{(Par)}$$

$$\frac{P' \equiv P \quad P \rightarrow Q \quad Q \equiv Q'}{P' \rightarrow Q'} \qquad \text{(Equiv)}$$

For any name a, the *observation predicate* $\downarrow_a$ denotes the possibility of a process immediately performing a communication with external environment along a. Thus, $P \downarrow_a$ holds if P has a sub-term $\overline{a}b$ or $a(\tilde{x}).Q$ which is not under the scope of a νa or of a $b(a)$. We write $P \rightarrow_{\mathsf{d}} P'$ if the reduction $P \rightarrow P'$ is *deterministic*, that is: $P \rightarrow P''$ implies $P' \equiv P''$, and there is no a such that $P \downarrow_a$.

5.3 Input/output types

The set type of *input/output types* is inductively defined as follows.

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

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 ($\mathsf{b}\langle\tilde{S}\rangle$).

Type environments, or *typings*, denoted by the Greek letters 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 relation $\vdash$, which can be found in reference [12]. IF X is the set $\{x_1, \ldots, x_n\}$, we write $X : S$ to mean the typing $x_1 : S, \ldots, x_n : S$.

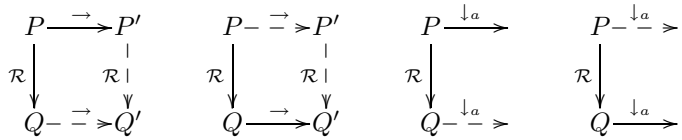
5.4 Contexts

A *process context* C is a process expression with a hole in it. We let context denote the set of contexts, and we write $C[P]$ for the expression obtained by filling the hole in C with P. The hole itself is denoted by []. Reduction is defined for processes, but can be easily extended to contexts by saying that $C \to C'$ whenever, $C[P] \to Q$ implies $Q \equiv C'[P]$, for P an arbitrary process.

To type contexts, we use context typings. Since a context is a process with a hole, we need two typings: one for the (process that fills the) hole, the other for the context (with the hole filled by the process). Using Δ for the typing of the hole, and Γ for that of the context, a *context-typing* is denoted by $\Gamma[\Delta]$. Then we write $\Gamma[\Delta] \vdash C$ when $\Delta \vdash P$ implies $\Gamma \vdash C[P]$. In this case, we say that C is a $\Gamma[\Delta]$-*context*.

5.5 Behavioural equality

Behavioural equality is based on the notion of barbed congruence [11]. The definition uses the reduction relation and the observation predicate defined earlier. A relation $\mathcal{R}$ is a *strong barbed bisimulation* if



Two processes P, Q are called *strongly barbed bisimilar* if $P\mathcal{R}Q$ for some strong barbed bisimulation $\mathcal{R}$. Barbed bisimulation by itself is a rather coarse relation; better discriminating power can be achieved by using the induced congruence, called *barbed congruence*. In typed calculi, the processes being compared must obey the same typing, and the contexts employed must be compatible with this typing. Suppose that processes P, Q obey a given typing Δ. We say that P, Q are *strongly barbed congruent at* Δ, written $P \sim_\Delta Q$, if, for all typings Γ, and all $\Gamma[\Delta]$-contexts C, we have that $C[P]$ and $C[Q]$ are strongly barbed bisimilar.

The weak version of barbed bisimulation is defined by replacing the reduction $\to$ with the relation $\to^*$, and the observation predicate $\downarrow_a$ with the predicate $\to^*\downarrow_a$. We write $\approx_\Delta$ for the *(weak) barbed congruence at* Δ.

Extending barbed congruence to contexts, we say that two $\Gamma[\Delta]$-contexts C, C' are strongly barbed congruent, and write $C \sim_{\Gamma[\Delta]} C'$, if $\Delta \vdash P$ implies $C[P] \sim_\Gamma C'[P]$, and similarly for the weak regime.

5.6 Replication theorems

The replication theorems constitute the cornerstone of results in the paper. All proofs ultimately, directly or indirectly, are based on these theorems taken from the literature.

We say that a, a name free in P, is used as a trigger in P, if P obeys a typing Γ, and the tag of the type of a in Γ is o.

Theorem 10 (Replication theorems [12, 12]). *If a is used as a trigger in P, Q, R, then*

1. $\nu a\ !a(\tilde{x}).Q \mid b(y).P \sim_\Delta b(y).(\nu a\ !a(\tilde{x}).Q \mid P)$;

2. $\nu a\ !a(\tilde{x}).P \mid !b(\tilde{y}).Q \sim_\Delta !b(\tilde{y}).(\nu a\ !a(\tilde{x}).P \mid Q)$.

3. $\nu a\ !a(\tilde{x}).Q \mid \nu b\ !b(\tilde{x}).Q \mid P \sim_\Delta \nu a\ !a(\tilde{x}).Q \mid P[a/b]$, *if* b *is not free in* Q *and is used as a trigger in* P.

Proof. The two first clauses are the first and second replication theorems [12]. The third clause constitutes part of the proof of Lemma C.2.2 in [12]. □

Relying on the variable convention, we omit the condition on clause 1 above: y is not free in Q since it is bound in $b(y).P$, and $a \neq b$ since b is free in the left hand side whereas a is bound.

Theorem 11 (Link theorem [16] B.12). *If* $\Delta \vdash \overline{a}b$ *where* $\Delta(a) = \mathsf{oo}S$ *for some* S, *then* $\overline{a}b \approx_\Delta \nu c\colon \mathsf{o}S\,\overline{a}c \mid !c(x).\overline{b}x$.

Chapter 6

Encoding the cbv-λ into π

We now introduce our encoding of the call-by-value λ-calculus into the π-calculus, thereby defining a machine that can be put into correspondence with lambda-machine defined in section 3. The section also presents a comparison with Milner's encoding, and shows that there is room for optimizing the encoding.

6.1 The encoding

To define the compilation scheme we need to know:

1. the λ-term to encode,
2. the channel where the λ-term is to be located (the location of the term), and
3. the scope of this channel, that is, the piece of code (a process) where this channel is to be visible.

The novelty with respect to Milner's encoding resides exactly in this last item. Instead of providing an encoding that may be placed in any context, we ask for the scope where the term is to run, and return a process that includes the encoding of the term and the scope. So we have a mapping

$$[\![M]\!]_a P : \mathsf{term} \to \mathsf{name} \to \mathsf{process} \to \mathsf{process}.$$

By using the contexts we can simplify the presentation and work with a mapping

$$[\![M]\!]_a : \mathsf{term} \to \mathsf{name} \to \mathsf{context}.$$

A form $[\![M]\!]_a$ represents a context in whose hole name a denotes the location of term M. The process in the hole is blocked (at some prefix $r(a).[]$) until the term attains its normal form. Name a is free in the hole, allowing the process in the hole to refer to (the value of) M.

We are now in position to present the encoding. Starting with λ-variables, and assuming that the set $\mathsf{variable}$ is an infinite and co-infinite subset of name, we have

$$[\![x]\!]_a \stackrel{\mathsf{def}}{=} [][x/a],$$

so that variables are not really translated, but else syntactically substituted in the hole. For example, $[\![y]\!]_a \overline{u}a \stackrel{\mathsf{def}}{=} \overline{u}y$.

For abstractions, we have

$$[\![\lambda x M]\!]_a \stackrel{\mathsf{def}}{=} \nu a \;\; !a(xr).[\![M]\!]_b \overline{r}b \mid [],$$

where names b and r are fresh. Notice that a, the location of the term, is local to the hole, so that the process placed in the hole "sees", or "knows", the abstraction. The part $!a(xr).[\![M]\!]_b \overline{r}b$ is called an *environment entry*; in the sequel we abbreviate such a process to $a := \lambda x M$. The location a of the term is replicated since the function may be invoked an arbitrary number of times from within the process in the hole. Channel a receives a pair of names: the first is the argument to the function, the second is a name intended to reply the result of the function, and is usually called a *reply-to name*. In this way, the function, when invoked, evaluates M (with x replaced by the actual argument) to a fresh name b, and returns this name in the reply-to name r supplied by the caller.

Finally, for applications, we have

$$[\![MN]\!]_a \stackrel{\mathsf{def}}{=} \nu r \; [\![M]\!]_b [\![N]\!]_c \overline{b}cr \mid r(a).[],$$

where names b, c, r are fresh. We first evaluate M to b, then N to c, and then invoke the function at b with the argument c and request to have the result sent to r. Finally we wait for the result at r, and instantiate it in the hole as a, since the process in the hole expects to see the result of the evaluation of MN located at a.

We have omitted the square brackets in the contexts above: instead of writing $[\![M]\!]_b[[\![N]\!]_c[\overline{b}cr]]$, we simply write $[\![M]\!]_b [\![N]\!]_c \overline{b}cr$, and similarly for abstractions.

The below examples help in understanding the dynamics of the encoding.

Example 12. 1. Values involve no reduction: λ-variables are syntactically substituted in the context; the location of an abstraction is local to the context of the abstraction . Let $\mathbf{I}$ denote the term $\lambda z z$. Then,

$$[\![\mathbf{I}]\!]_a \stackrel{\mathsf{def}}{=} \nu a \;\; !a(zr).\overline{r}z \mid [].$$

Notice that there is no reduction involved in letting the hole know the location of $\mathbf{I}$.

2. An application of the beta-axiom corresponds to a pair of π-reductions, one to invoke the function, the other to get the reply. In the sequel we underline each redex.

$$\begin{aligned}
[\![\mathbf{I}x]\!]_a &\stackrel{\mathsf{def}}{=} \\
\nu r b \;\; \underline{!b(xr').\overline{r'}x} \mid \underline{\overline{b}xr} \mid r(a).[] &\to_{\mathsf{d}} \\
\nu r b \, b := \mathbf{I} \mid \underline{\overline{r}x} \mid \underline{r(a).[]} &\to_{\mathsf{d}} \\
(\nu b \, b := \mathbf{I}) \mid []\{x/a\} &\sim \\
[]\{x/a\} &\stackrel{\mathsf{def}}{=} [\![x]\!]_a
\end{aligned}$$

The last step represents the garbage collection of an environment entry that is no longer accessible.

3. Reduction is determinate. Below the leftmost redex ($\mathbf{I}x$) must be reduced before the rightmost one ($\mathbf{I}y$), since the latter is under the prefix $r'(b)$. Using the above result twice, we have the following reduction.

$$\begin{aligned}
[\![\mathbf{I}x(\mathbf{I}y)]\!]_a &\stackrel{\mathsf{def}}{=} \\
\nu r r' \, [\![\mathbf{I}]\!]_d [\![x]\!]_e \overline{d} e r' \mid r'(b).([\![\mathbf{I}y]\!]_c \overline{b} c r \mid r(a).[]) &\to^2_{\mathsf{d}} \sim \\
\nu r \, [\![\mathbf{I}y]\!]_c \overline{x} c r \mid r(a).[] &\to^2_{\mathsf{d}} \sim \\
\nu r \, \overline{x} y r \mid r(a).[] &\stackrel{\mathsf{def}}{=} [\![xy]\!]_a
\end{aligned}$$

We can see that, on the λ-calculus side, we have

$$\mathbf{I}x(\mathbf{I}y) \to_{\mathsf{v}} x(\mathbf{I}y) \to_{\mathsf{v}} xy,$$

and that, on the π-calculus side, there are contexts C, C' such that

$$[\![\mathbf{I}x(\mathbf{I}y)]\!]_a \to^2_{\mathsf{d}} C \sim [\![x(\mathbf{I}y)]\!]_a \to^2_{\mathsf{d}} C' \sim [\![xy]\!]_a.$$

That is not always the case, as the next example witnesses.

4. Take for M the term $(\lambda x \mathbf{I}x)y$. We know that $M \to_{\mathsf{v}} \mathbf{I}y$, but there is no context C such that $[\![M]\!]_a \to^* C$ with $C \sim [\![\mathbf{I}y]\!]_a$. In fact, M has two nested redexes. As such, after two π-steps, the two function calls have been performed, but the two "returns" ($\overline{r}'y$ and $\overline{r}b$) are pending.

$$[\![M]\!]_a \to^2_{\mathsf{d}} \sim \nu r' \, \overline{r}' y \mid \nu r \, r'(b).\overline{r}b \mid r(a).[]$$

At this point there remains a "stack" with two "activation records"; we can easily see that two deterministic steps are what it takes to pop this stack, reducing the above context to $[\![y]\!]_a$. Compare with Example 9, where the corresponding SECD machine proceeds by performing two call operations followed by two return operations.

5. The last example focuses on a divergent reduction. Let $\Omega = \mathbf{Z}\mathbf{Z}$, and $\mathbf{Z} = \lambda zzz$. Then we have:

$$\begin{aligned}
[\![\Omega]\!]_a &\stackrel{\mathsf{def}}{=} \\
\nu r b \, \underline{b := \mathbf{Z}} \mid \nu c \, c := \mathbf{Z} \mid \underline{\overline{b} c r} \mid r(a).[] &\to_{\mathsf{d}} \sim \\
\nu r r' c \, \underline{c := \mathbf{Z}} \mid \underline{\overline{c} c r'} \mid r'(a).\overline{r}a \mid r(a).[] &\to_{\mathsf{d}} \\
\nu r r' r'' c \, \underline{c := \mathbf{Z}} \mid \underline{\overline{c} c r''} \mid r''(a).\overline{r'}a \mid r'(a).\overline{r}a \mid r(a).[] &\to_{\mathsf{d}} \\
\nu r r' r'' r''' c \, \underline{c := \mathbf{Z}} \mid \underline{\overline{c} c r'''} \mid r'''(a).\overline{r''}a \mid r''(a).\overline{r'}a \mid r'(a).\overline{r}a \mid r(a).[]
\end{aligned}$$

Notice the ever growing chain of forwarders (or activation records): $r'''(a).\overline{r''}a \mid r''(a).\overline{r'}a \mid r'(a).\overline{r}a$. Each β-reduction remains half-done: the function $\mathbf{Z}$ has been invoked; but there is a forwarder waiting to convey the result to the corresponding context (or, using the activation stack analogy, there remains an activation record to pop).

We conclude this section by analyzing the free names present in the encoding. The encoding introduces no free names other than the free variables in M. Also, any process placed in the hole of $[\![M]\!]_a$ may access the (encoding of) M via name a. This name, however, is not free in the context when filled up with a process.

Lemma 13. *1.* $\mathrm{fn}([\![M]\!]_a) = \mathrm{fv}(M)$*;*

2. $\mathrm{fn}([\![M]\!]_a P) = \mathrm{fv}(M) \cup \mathrm{fn}(P) - \{a\}$.

6.2 Typing the encoding

Encoded λ-terms are meant to run in a concurrent environment. Such environments are in general overly permissive. The terms, as we have encoded them, are somewhat sensible — they only behave correctly when placed in controlled surroundings.

Let us see what happens if we allow the placing of arbitrary processes in the hole of the encoding. Take the two terms **KI** and $\lambda y\mathbf{I}$.[1] We know that the two terms are β-equivalent (in fact the former reduces to the latter), but there is a process capable of distinguishing them. We have

$$[\![\mathbf{KI}]\!]_a \to^2_{\mathsf{d}} \sim \nu c\, c := \mathbf{I} \mid \nu a\, !a(yr).\overline{r}c \mid [] \quad (\stackrel{\mathsf{def}}{=} C_1)$$
$$[\![\lambda y\mathbf{I}]\!]_a \stackrel{\mathsf{def}}{=} \nu a\, !a(yr).(\nu c\, c := \mathbf{I} \mid \overline{r}c) \mid [] \quad (\stackrel{\mathsf{def}}{=} C_2)$$

By concentrating on the processes $!a(yr)\ldots$, we see that in the case of **KI** all invocations return the same name c, whereas for $\lambda y\mathbf{I}$ each invocation provides a fresh name. (Notice that although the names are different, the processes the names locate are syntactically equal.) How can we take distinguish the two cases? We invoke twice the function to obtain two names. If they are the same, then we are in presence of **KI**, otherwise we have $\lambda y\mathbf{I}$. All that remains is to compare two names for equality: we write on one of them, read from the other, and arrange for an observable action to emerge from the contractum. So, for the observer take the process

$$P \stackrel{\mathsf{def}}{=} \overline{a}xr \mid r(c').(\overline{a}xr \mid r(c'').(c'(xr).\overline{o} \mid \overline{c}''xr)).$$

We can easily check that

$$\begin{aligned}
[\![\mathbf{KI}]\!]_a P \to^6_{\mathsf{d}} & C_1[\underline{c(xr).\overline{o}} \mid \underline{\overline{c}xr}] \to_{\mathsf{d}} \\
& C_1[\overline{o}] \downarrow_o \\
[\![\lambda y\mathbf{I}]\!]_a P \to^4_{\mathsf{d}} & C_2[\nu c'\, c' := \mathbf{I} \mid \nu c''\, \underline{c'' := \mathbf{I}} \mid c'(xr).\overline{o} \mid \underline{\overline{c}''xr}] \to_{\mathsf{d}} \\
& C_2[\nu c'\, c' := \mathbf{I} \mid \nu c''\, c'' := \mathbf{I} \mid c'(xr).\overline{o} \mid \overline{r}c''] \not\to \not\downarrow_o
\end{aligned}$$

Only by violating the (hitherto implicit) protocol for locations of terms were we able to tell the two terms apart. In fact we have installed a reader $(c'(xr).\overline{o})$ on the location of a function.

What kind of processes may we place in the hole of an encoding? Processes that follow a discipline of i/o-types [12]. There are two kinds of names involved in the encoding:

1. *value names* comprising variables x, y, z, as well as locations of abstractions a, b, c; and
2. and *reply-to names* r, r', r'', names that return value names.

Let us call Val the type of the value names. How does this type look like? Looking at the fragment $[\![M]\!]_b[\![N]\!]_c\overline{b}cr$ of the encoding of MN, we know that Val must be of the form $\mathsf{o}\langle\mathsf{Val}, S\rangle$, for some type S. How does S looks like?

[1] $\mathbf{K}$ is the term λxyx; **KI** and $\lambda y\mathbf{I}$ are the terms used in Sangiorgi's thesis [15] for this exact purpose.

Concentrating on the part $[\![M]\!]_b\overline{r}b$ of the encoding of λxM, we easily conclude that S is the type oVal. So we have[2]

$$\mathsf{Val} \stackrel{\mathsf{def}}{=} \mu X.\mathsf{o}\langle X, \mathsf{o}X\rangle.$$

In the encoding of an abstraction, the name a locating the abstraction may be used for input (at $!a(xr)\ldots$) as well as for output (from the process within the hole). Such a name is of type

$$\mathsf{Val}^{\mathsf{b}} \stackrel{\mathsf{def}}{=} \mathsf{b}\langle \mathsf{Val}, \mathsf{o}\,\mathsf{Val}\rangle.$$

Also, in the encoding of an application, the reply-to name r is used to output the value of the application (in $[\![M]\!]_b[\![N]\!]_c\overline{b}cr$), as well as to input (in $r(a).[]$).

The encoding, where bound names are annotated with types, is as follows.

$$\begin{aligned}
[\![x]\!]_a &\stackrel{\mathsf{def}}{=} [][x/a] \\
[\![\lambda xM]\!]_a &\stackrel{\mathsf{def}}{=} \nu a\colon \mathsf{Val}^{\mathsf{b}}\ !a(x\colon \mathsf{Val}, r\colon \mathsf{o}\,\mathsf{Val}).[\![M]\!]_b\overline{r}b \mid [] \\
[\![MN]\!]_a &\stackrel{\mathsf{def}}{=} \nu r\colon \mathsf{b}\,\mathsf{Val}\,[\![M]\!]_b[\![N]\!]_c\overline{b}cr \mid r(a\colon \mathsf{Val}).[]
\end{aligned}$$

The main result related to the typing of the encoding states that the free variables in a term all have type Val, and so does the location of the encoded term.

Lemma 14. *If* $\mathrm{fv}(M) : \mathsf{Val} \subseteq \Gamma$*, then* $\Gamma[\Gamma, a : \mathsf{Val}] \vdash [\![M]\!]_a$*.*

Proof. By induction on the structure of M, using basic results of the type system such as weakening and substitution [12]. □

6.3 Convergence correspondence

Lemma 14 says that when Δ is a typing assigning type Val to all the free variables in a term M, then $[\![M]\!]_a$ is a $\Delta[\Delta, a : \mathsf{Val}]$-context. In the sequel we abbreviate context-typings of the form $\Delta[\Delta, a : \mathsf{Val}]$ to Δ, so that, when we write $[\![M]\!]_a \sim_\Delta [\![N]\!]_a$, we mean $[\![M]\!]_a \sim_{\Delta[\Delta,a:\mathsf{Val}]} [\![N]\!]_a$ where Δ contains the typing $\mathrm{fn}(MN) : \mathsf{Val}$.

The pi-based machine we are interested at, the *context machine*, $\mathsf{Context}$, is the triple $\langle \mathsf{context}, \rightarrow_{\mathsf{d}}, \sim_\Delta\rangle$. Fix a name a not in $\mathsf{variable}$; the $\mathsf{term\text{-}to\text{-}context}$ *correspondence* is defined as

$$\mathsf{term\text{-}to\text{-}context}(M) \stackrel{\mathsf{def}}{=} [\![M]\!]_a \qquad \text{if } M \text{ is closed}$$

Theorem 15. *If* $M \downarrow^n_{\mathsf{v}} N$*, then* $[\![M]\!]_a \downarrow^{2n}_{\mathsf{d}} \sim_\Delta [\![N]\!]_a$*.*

Theorem 16. *If* $[\![M]\!]_a \downarrow^n C$*, then* n *is even, and there is a term* N *such that* $M \downarrow^{n/2}_{\mathsf{v}} N$ *with* $C \sim_\Delta [\![N]\!]_a$*.*

Corollary 17. *The* $\mathsf{term\text{-}to\text{-}context}$ *correspondence is convergent.*

[2] This is exactly the type of value names in Milner's encoding [12], cf. section 7.

From Proposition 5 it follows that the encoding is adequate for the call-by-value lambda calculus. The proofs of the two Theorems above require two lemmas.

Lemma 18 (Value exchange lemma). $[\![M]\!]_a[\![V]\!]_b \sim_\Delta [\![V]\!]_b[\![M]\!]_a$.

Proof. Case analysis on V; case analysis on M when V is an abstraction. Uses the replication theorem 10.1. First notice that, since term locations are taken freshly, a does not occur free in V, and b does not occur free in M.

Case V is a variable. Notice that, by hypothesis, b is not free in M.

$$[\![M]\!]_a[\![V]\!]_b \stackrel{\mathsf{def}}{=} ([\![M]\!]_a)[V/b] \stackrel{\mathsf{def}}{=} [\![V]\!]_b[\![M]\!]_a$$

Case V is an abstraction. The difficult case is when is M an application; the reader is invited to check the two remaining cases.

$$\begin{aligned}
[\![NL]\!]_a[\![V]\!]_b &\stackrel{\mathsf{def}}{=} \\
\nu r\, [\![N]\!]_c[\![L]\!]_d\overline{c}dr \mid r(a).(\nu b\, b := V \mid []) &\sim_\Delta \qquad \text{(Replication theorem 10.1)} \\
\nu r\, [\![N]\!]_c[\![L]\!]_d\overline{c}dr \mid \nu b\, b := V \mid r(a).[] &\equiv \\
\nu b\, b := V \mid \nu r\, [\![N]\!]_c[\![L]\!]_d\overline{c}dr \mid r(a).[] &\stackrel{\mathsf{def}}{=} \\
[\![V]\!]_b[\![NL]\!]_a &
\end{aligned}$$

In the application of the structural congruence, notice that r is not free in $b := V$ since it is taken freshly, by definition. □

Lemma 19 (Substitution lemma). *If x is not free in the hole, then* $[\![V]\!]_x[\![M]\!]_a \sim_\Delta [\![M[V/x]]\!]_a$.

Proof. Case analysis on V; induction on M when V is an abstraction. Uses the replication theorems 10.2 and 10.3, and the exchange lemma 18.

Case V is a variable. The two sides of the equation become syntactically equal. The remaining cases are for when V is an abstraction.

Case M is the variable x. Notice that, by hypothesis, x is not free in the hole.

$$[\![V]\!]_a \stackrel{\mathsf{def}}{=} \nu a\, a := V \mid [] \equiv_\alpha \nu x\, x := V \mid [][x/a] \stackrel{\mathsf{def}}{=} [\![V]\!]_x[\![x]\!]_a$$

Case M is a variable $x \neq y$. Again notice that x is not free in the hole.

$$[\![y]\!]_a \stackrel{\mathsf{def}}{=} [][y/a] \sim \nu x\, x := V \mid [][y/a] \stackrel{\mathsf{def}}{=} [\![V]\!]_x[\![y]\!]_a$$

The $\sim$-step represents the generation of a garbage environment entry.

Case M is an abstraction λyN.

$$\begin{aligned}
[\![\lambda yN[V/x]]\!]_a &\stackrel{\mathsf{def}}{=} \\
\nu a\, !a(yr).[\![N[V/x]]\!]_b\overline{r}b \mid [] &\sim_\Delta \qquad \text{(Induction)} \\
\nu a\, !a(yr).[\![V]\!]_x[\![N]\!]_b\overline{r}b \mid [] &\sim_\Delta \qquad \text{(Replication theorem 10.2)} \\
[\![V]\!]_x(\nu a\, !a(yr).[\![N]\!]_b\overline{r}b \mid []) &\stackrel{\mathsf{def}}{=} \\
[\![V]\!]_x[\![\lambda yN]\!]_a &
\end{aligned}$$

Case M is an application NL.

$$\begin{aligned}
[\![N[V/x]L[V/x]]\!]_a &\stackrel{\mathsf{def}}{=} \\
\nu r\, [\![N[V/x]]\!]_b [\![L[V/x]]\!]_c \overline{b}cr \mid r(a).[] &\sim_\Delta && \text{(Induction twice)} \\
\nu r\, [\![V]\!]_x [\![N]\!]_b [\![V]\!]_x [\![L]\!]_c \overline{b}cr \mid r(a).[] &\equiv_\alpha && (y \text{ fresh}) \\
\nu r\, [\![V]\!]_x [\![N]\!]_b [\![V]\!]_y [\![L[y/x]]\!]_c \overline{b}cr \mid r(a).[] &\sim_\Delta && \text{(Exchange lemma 18)} \\
\nu r\, [\![V]\!]_x [\![V]\!]_y [\![N]\!]_b [\![L[y/x]]\!]_c \overline{b}cr \mid r(a).[] &\sim_\Delta && \text{(Replication theorem 10.3)} \\
\nu r\, [\![V]\!]_x [\![N]\!]_b [\![L]\!]_c \overline{b}cr \mid r(a).[] &\stackrel{\mathsf{def}}{=} \\
[\![V]\!]_x [\![NL]\!]_a
\end{aligned}$$

□

Proof of Theorem 15. By induction on n; uses the Substitution lemma 19. When n is zero, $M = N$, and we are done. When n is positive, M is an application M_1M_2. Let $M_1 \downarrow_{\mathsf{v}}^{n_1} \lambda xL$, $M_2 \downarrow_{\mathsf{v}}^{V} V$, and $L[V/x] \downarrow_{\mathsf{v}}^{n_3} N$, with $n = n_1+n_2+n_3+1$. Then we have

$$\begin{aligned}
[\![M_1M_2]\!]_a &\stackrel{\mathsf{def}}{=} \\
\nu r\, [\![M_1]\!]_b [\![M_2]\!]_c \overline{b}cr \mid r(a).[] &\to_{\mathsf{d}}^{2(n_1+n_2)} \sim && \text{(Induction hypothesis)} \\
\nu r\, \nu b\, \underline{b := \lambda xl} \mid \nu c\, c := V \mid \underline{\overline{b}cr} \mid r(a).[] &\to_{\mathsf{d}} \sim && \text{(Garbage collection)} \\
\nu r\, \nu c\, c := V \mid [\![L[c/x]]\!]_d \overline{r}d \mid r(a).[] &\equiv_\alpha \stackrel{\mathsf{def}}{=} \\
\nu r\, [\![V]\!]_x [\![L]\!]_d \overline{r}d \mid r(a).[] &\sim_\Delta && \text{(Substitution lemma 19)} \\
\nu r\, [\![L[V/x]]\!]_d \overline{r}d \mid r(a).[] &\to_{\mathsf{d}}^{2n_3} \sim_\Delta && \text{(Induction hypothesis)} \\
\nu r\, [\![N]\!]_d \overline{r}d \mid r(a).[] &\stackrel{\mathsf{def}}{=} \\
\nu r\, \nu d\, d := N \mid \underline{\overline{r}d} \mid \underline{r(a).[]} &\to_{\mathsf{d}} \equiv_\alpha \\
\nu a\, a := N \mid [] &\stackrel{\mathsf{def}}{=} \\
[\![N]\!]_a
\end{aligned}$$

□

Proof of Theorem 16. By induction on n; uses the Substitution lemma 19. When n is zero, $[\![M]\!]_a = C$, and we are done. When n is positive, M is an application M_1M_2. Let $[\![M_1]\!]_b \downarrow_{\mathsf{d}}^{n_1} \sim_\Delta [\![\lambda xL]\!]_b$, $[\![M_2]\!]_c \downarrow_{\mathsf{d}}^{n_2} \sim_\Delta [\![V]\!]_c$, and $[\![V]\!]_c [\![L[c/x]]\!]_d \downarrow_{\mathsf{d}}^{n_3} \sim_\Delta [\![N]\!]_d$, with $n = n_1 + n_2 + n_3 + 2$. Then, noticing that by the substitution lemma 19, $[\![V]\!]_c [\![L[c/x]]\!]_d \sim_\Delta [\![L[V/x]]\!]$, by induction we have

$$M_1M_2 \to_{\mathsf{v}}^{n_1/2} (\lambda xL)M_2 \to_{\mathsf{v}}^{n_2/2} (\lambda xL)V \to_{\mathsf{v}} L[V/x] \to_{\mathsf{v}}^{n_3/2} N$$

□

6.4 Weak operational correspondence

The term-to-context correspondence is not operational; Example 12.4 provides a counter-example. Nonetheless, if we compare lambda and pi reduction steps up to weak barbed congruence, we get a form of weak operational correspondence.

Theorem 20.

As a corollary we get that the encoding is valid for the call-by-value lambda-calculus.

Corollary 21 (Validity). *If* $M =_\beta N$*, then* $[\![M]\!]_a \approx_\Delta [\![N]\!]_a$.

The proof of Theorem 20 requires the Substitution Lemma 19, and three further lemmas. The first says that there is an alternative encoding for applications.

Lemma 22 (Alternative encoding). $[\![MN]\!]_a \sim_\Delta [\![M]\!]_b[\![N]\!]_c[\nu r\, \overline{b}cr \mid r(a).[]]$.

Proof. Apply the replication theorem 10.1 once for each application in MN, to obtain

$$[\![MN]\!]_a \sim_\Delta \nu r\, [\![M]\!]_b[\![N]\!]_c[\overline{b}cr \mid r(a).[]].$$

Finally, since reply-to names are taken freshly, use structural congruence to bring the νr inwards, and obtain the result. □

The forwarder lemma accounts for the presence of the weak congruence in the main result of this section.

Lemma 23 (Forwarder lemma). $\nu r\, [\![M]\!]_b\overline{r}b \mid r(a).[] \approx_\Delta [\![M]\!]_a$.

Proof. A case analysis on M. Uses the link theorem 11, and the alternative encoding 22.

Case M is a value V.

$$\nu r\, [\![M]\!]_b\overline{r}b \mid r(a).[] \to_{\mathsf{d}} [\![M]\!]_a$$

Case M is an application NL.

$$\begin{aligned}
\nu r\, [\![NL]\!]_b\overline{r}b \mid r(a).[] &\sim_\Delta && \text{(Alternative encoding 22)}\\
\nu r\, [\![N]\!]_c[\![L]\!]_d[\nu r'\, \overline{c}dr' \mid r'(b).\overline{r}b) \mid r(a).[]] &\approx_\Delta && \text{(Link theorem 11)}\\
\nu r\, [\![N]\!]_c[\![L]\!]_d\overline{c}dr \mid r(a).[] &\stackrel{\mathsf{def}}{=}\\
[\![NL]\!]_a
\end{aligned}$$

Unlike theorem 11, the prefix $r(a).[]$ is not replicated, due to the linearity constraint on reply-to names. □

A β-step corresponds to a deterministic pi-step ($\to_{\mathsf{d}}$), the calling of the function, possibly followed by a return step, accounted for by the barbed congruence.

Lemma 24 (Beta lemma). $[\![(\lambda xM)V]\!]_a \to_{\mathsf{d}} \approx_\Delta [\![M[V/x]]\!]_a$.

Proof. Case analysis on V. Uses the substitution lemma 19 and the forwarder lemma 23.

Case V is the variable.

$$\begin{aligned}
[\![(\lambda x M)V]\!]_a &\stackrel{\mathsf{def}}{=} \\
\nu r b\, \underline{b := \lambda x M} \mid \underline{\overline{b} V r} \mid r(a).[] &\to_{\mathsf{d}} \\
(\nu b\, b := \lambda x M) \mid \nu r\, [\![M[V/x]]\!]_d \overline{r} d \mid r(a).[] &\sim && \text{(Garbage collection)} \\
\nu r\, [\![M[V/x]]\!]_d \overline{r} d \mid r(a).[] &\approx_\Delta && \text{(Forwarder lemma 23)} \\
[\![M[V/x]]\!]_a &
\end{aligned}$$

Case V is an abstraction.

$$\begin{aligned}
[\![(\lambda x M)V]\!]_a &\stackrel{\mathsf{def}}{=} \\
\nu r b\, \underline{b := \lambda x M} \mid \nu c\, c := V \mid \underline{\overline{b} c r} \mid r(a).[] &\to_{\mathsf{d}} \\
(\nu b\, b := \lambda x M) \mid \nu c\, c := V \mid \nu r\, [\![M[c/x]]\!]_d \overline{r} d \mid r(a).[] &\sim && \text{(Garbage collection)} \\
\nu c\, c := V \mid \nu r\, [\![M[c/x]]\!]_d \overline{r} d \mid r(a).[] &\approx_\Delta && \text{(Forwarder lemma 23)} \\
\nu c\, c := V \mid [\![M[c/x]]\!]_a &\sim_\Delta && \text{(Substitution lemma 19)} \\
[\![M[V/x]]\!]_a &
\end{aligned}$$

In the application of the substitution lemma above, notice that x, being bound in M, is not free in the hole by the variable convention. □

Proof of Theorem 20. For the first diagram, use induction on the structure of reduction. When the last rule is the β-rule, use the beta lemma 24; for the two remaining cases use a straightforward induction.

For the second diagram, we know that, if C reduces, then M is an application. Since M is closed, there is a term M' such that $M \to_{\mathsf{v}} M'$. Using the first diagram, we get $C \to_{\mathsf{d}} \approx_\Delta [\![M']\!]_a$. □

6.5 Further optimizations

So we manage to mimic a β-reduction step with two π-steps. Is is this best we can do? For general terms, the author believes so. As long as we compile lambda-terms in given pi-locations, the interaction with the term must be via this location. In particular, invoking a function needs a message exchange at the function's location. Then, if we care for the result, we must read it somehow; I cannot see any other way than exchanging a message for this purpose.

This is not to say that in particular cases we cannot do better. As an example, we try to optimize recursive functions. Take a *call-by-value fixed-point combinator* [2].

$$\mathbf{Z} \stackrel{\mathsf{def}}{=} \lambda f \mathbf{W}\mathbf{W} \qquad \text{with} \qquad \mathbf{W} \stackrel{\mathsf{def}}{=} \lambda x \lambda y f(xx)y.$$

The reader can easily check that, for each abstraction $\lambda z M$, there is a term $\mathbf{Z}'$ such that,

$$\mathbf{Z}(\lambda z M) \to_{\mathsf{v}} \mathbf{Z}' \qquad \text{and} \qquad \mathbf{Z}' V \to_{\mathsf{v}}^3 M[\mathbf{Z}'/z]V$$

Then, using the substitution lemma 19, we have that

$$[\![\mathbf{Z}'V]\!]_a \to_{\mathsf{d}}^{5} \sim_{\Delta} \nu r\, [\![M[\mathbf{Z}'/z]V]\!]_b \overline{r} b \mid r(a).[].$$

We thus see that the generic encoding requires five-steps (plus the pending reduction at r; cf. theorem 17) for the function to access its fixed-point combinator $\mathbf{Z}'$. We propose to compile recursive functions by letting the body of the function know its location, thus allowing for the function to call itself directly through its location,

$$[\![\mathbf{Z}(\lambda z M)]\!]_a \stackrel{\mathsf{def}}{=} [\![M[a/z]]\!]_a.$$

The correctness of this optimization remains open at the time of this writing.

Chapter 7

Comparison with Milner's encoding

Milner proposed two encodings for the call-by-value lambda-calculus [8]. The correctness of one of the encodings is proved in the paper; the analysis of the other (more efficient, the "light version") was established later by Pierce and Sangiorgi [12]. This last encoding is quite similar to ours: not only value names and reply-to names share the same types, but the two encodings are barbed congruent.

7.1 The encoding

Milner's encoding is a mapping $\{\!|\, M \,|\!\}_a : \mathsf{term} \to \mathsf{name} \to \mathsf{process}$, defined as

$$\begin{aligned} \{\!|\, x \,|\!\}_p &\stackrel{\mathsf{def}}{=} \overline{p}x \\ \{\!|\, \lambda x M \,|\!\}_p &\stackrel{\mathsf{def}}{=} \nu a \;\; !a(xq).\{\!|\, M \,|\!\}_q \mid \overline{p}a \\ \{\!|\, MN \,|\!\}_p &\stackrel{\mathsf{def}}{=} \nu q \; \{\!|\, M \,|\!\}_q \mid q(a).(\nu r \; \{\!|\, N \,|\!\}_r \mid r(b).\overline{a}bp) \end{aligned}$$

where names a, b, c, q, r are always fresh. As before we abbreviate processes of the form $!a(xq).\{\!|\, M \,|\!\}_q$ to $a := \lambda x M$, relying on the surroundings to select the appropriate definition for an environment entry.

First, notice that values write their value-names (either the variable itself, or the location of the abstraction) on some reply-to name. An environment entry gets the argument to the function and the channel where to locate the result; if the evaluation of the function converges (to some value) then this channel is written (with the value). In order to mimic an application MN located at p, we first evaluate the function M on some fresh reply-to name q, and wait for the result b on q. Then we do the same for the argument N, thus obtaining its result c. All that remains is to invoke b with c, requesting for the resulting value to be sent on p.

The *process machine*, $\mathsf{Process}$, is given by $\langle \mathsf{process}, \to_{\mathsf{d}}, \sim \rangle$, where all three components are defined in section 5. Fix a name a not in $\mathsf{variable}$; the $\mathsf{term\text{-}to\text{-}}$

process *correspondence* is the the function defined as

$$\mathsf{term\text{-}to\text{-}process}(M) \stackrel{\mathsf{def}}{=} \{\!|\, M \,|\!\}_a \qquad \text{if } M \text{ is closed}$$

This correspondence is discussed by Sangiorgi [16].

Theorem 25. *The* term-to-process *correspondence is an operational correspondence.*

Proof. The two first diagrams are corollaries 5.19 and 5.14 in Sangiorgi [16]; the third constitutes a simple exercise. □

7.2 Comparing

For comparison purposes, we are interested in the following theorem.

Theorem 26 ([16], Corollary 5.19). *For M closed, $M \to_{\mathsf{v}} N$ implies $\{\!|\, M \,|\!\}_p \to^3_{\mathsf{d}} \sim_{p:\,\mathsf{oVal}} \{\!|\, N \,|\!\}_p$.*

Notice that since M is closed, the only free name in $\{\!|\, M \,|\!\}_p$ is p, and p is a reply-to name; hence the typing $p\colon \mathsf{oVal}$ in the congruence above.

Milner's encoding enjoys a simple property: one reduction step in the call-by-value λ-calculus corresponds to three (deterministic) reduction steps in the π-calculus, modulo strong barbed congruence (theorem 26). In our case, all we can say is that a normalizing term takes, on average, two pi-steps to simulate a beta-step (theorem 17). Put in another way, Milner's encoding is operational (Theorem 25), whereas our is only convergent (Corollary 17).

The two encodings cannot be directly compared, for they have different signatures: Milner's expects a term and a name, whereas ours further expects a process. A simple analysis of the two encodings, leads us to compare the the following pairs of processes and contexts

$$\begin{array}{rcl} \{\!|\, M \,|\!\}_p & \text{and} & [\![M]\!]_a \overline{p} a, \\ [\![M]\!]_a & \text{and} & \nu p\, \{\!|\, M \,|\!\}_p \mid p(a).[]. \end{array}$$

Now we can ask: are $\{\!|\, M \,|\!\}_a$ and $[\![M]\!]_a \overline{p} a$ bisimilar? Not in general. In an untyped scenario, we can find a process able to distinguish the two encodings.[1] However, if we stick to typed contexts, we can show that the two encodings are indeed bisimilar.

Theorem 27. *For M closed,*

1. $\{\!|\, M \,|\!\}_p \approx_{p:\,\mathsf{oVal}} [\![M]\!]_a \overline{p} a$;
2. $[\![M]\!]_a P \approx_\emptyset \nu p\, \{\!|\, M \,|\!\}_p \mid p(a).P$.

Proof. 1. By induction on the structure of M. Uses the main results 20, 26 of the two encodings.

Case M is a variable.

$$\{\!|\, M \,|\!\}_p \stackrel{\mathsf{def}}{=} \overline{p} M \stackrel{\mathsf{def}}{=} [\![M]\!]_a \overline{p} a$$

[1] To distinguish the two encodings of the term $x\mathbf{I}$ take the process $x(bq).(\overline{p}d \mid q(c).\overline{c}er)$. Notice, however that neither of the encodings was designed to run on arbitrary contexts.

Case M is an abstraction $\lambda x N$.

$$\begin{aligned} \{\!|\, M \,|\!\}_p &\stackrel{\mathsf{def}}{=} \\ \nu a\ !a(xq).\{\!|\, N \,|\!\}_q \mid \overline{p}a &\approx_{p:\,\mathsf{oVal}} \qquad \text{(Induction)} \\ \nu a\ !a(xq).[\![N]\!]_b \overline{q}b \mid \overline{p}a &\stackrel{\mathsf{def}}{=} \\ [\![M]\!]_a \overline{p}a & \end{aligned}$$

Case M is an application. Since M is closed, there is a term N such that $M \to_{\mathsf{v}} N$. By the main result 26 of Milner's encoding, we have

$$[\![M]\!] \approx_{p:\,\mathsf{oVal}} \{\!|\, N \,|\!\}_p.$$

By the main result 20 of our encoding,

$$[\![M]\!]_a \overline{p}a \approx_{p:\,\mathsf{oVal}} [\![N]\!]_a \overline{p}a.$$

The result follows by induction.

2. Similar. □

Chapter 8

Encoding the SECD machine into π

We now present our encoding of the SECD machine into the π-calculus, thereby defining a correspondence that turns out to be operational.

8.1 The encoding

To reduce the number of rules in the translation, we write ME do denote the closure of a term M in an environment E, and define it as follows.

$$xE \stackrel{\mathsf{def}}{=} E(x) \qquad ME \stackrel{\mathsf{def}}{=} \langle M, E\rangle \quad (M \notin \mathsf{variable})$$

Closures and environments are translated into pi-contexts. More precisely, we define a mapping $[\![\mathrm{Cl}]\!]_a$: $\mathsf{closure} \to \mathsf{name} \to \mathsf{context}$ that, given a closure and a name, returns a context in whose hole the name denotes the location of the term in the closure. The mapping for environments is $[\![E]\!]_a$: $\mathsf{environment} \to \mathsf{name} \to \mathsf{context}$, that, given an environment E, yields a context where each term M_i is located at a name x_i, if $\langle x_i, \mathrm{Cl}_i\rangle$ is a closure in E. The two mappings are mutually recursive.

$$[\![\langle M, E\rangle]\!]_a \stackrel{\mathsf{def}}{=} [\![E]\!][\![M]\!]_a \qquad [\![\emptyset]\!] \stackrel{\mathsf{def}}{=} [] \qquad [\![E\{\mathrm{Cl}/x\}]\!] \stackrel{\mathsf{def}}{=} [\![\mathrm{Cl}]\!]_x[\![E]\!]$$

An important property of the above encoding is that if a closure Cl contains only values, then the hole in $[\![\mathrm{Cl}]\!]_a$ is under no prefix. We say that

1. a closure $\langle M, E\rangle$ is a *value closure* if E is a value environment;
2. an environment E is a *value environment* if, for every variable x in the domain of E, $E(x)$ is a value closure.
3. nil is a *value dump*; if the closures in S are value closures, each $\langle M, E\rangle$ for M in C is a value closure, and D is a value dump, then $\langle S, E, C, D\rangle$ is a *value dump.*

Proposition 28. *If Cl is a value closure, then the hole in $[\![Cl]\!]_a$ is under no prefix.*

Proof. By expanding the definitions of $[\![V]\!]_a$ we see that the hole in $[\![E]\!][\![V]\!]_a$ is under no prefix if the hole in $[\![E]\!]$ is under no prefix. To show the that the hole in $[\![E]\!] = []$ is under no prefix, we proceed by induction. Clearly the hole in $[\![\emptyset]\!]$ is under no prefix; as for $[\![E\{\mathrm{Cl}/x\}]\!] = [\![\mathrm{Cl}]\!]_x[\![E]\!]$, by induction the holes in $[\![\mathrm{Cl}]\!]_x$ and in $[\![E]\!]$ are under no prefix. □

The mapping $[\![D]\!]_{\vec{a}} : \mathsf{dump} \to \mathsf{name}^+ \to \mathsf{context}$ is inductively defined by the following rules that must be tried from top to bottom.

$$[\![\langle S, E, N : M : \mathsf{ap} : C, D\rangle]\!]_{\vec{a}} \stackrel{\mathsf{def}}{=} [\![NE]\!]_b[\![\langle S, E, M : \mathsf{ap} : C, D\rangle]\!]_{b\vec{a}} \qquad (\textsc{Ctr2})$$

$$[\![\langle S, E, M : \mathsf{ap} : C, D\rangle]\!]_{b\vec{a}} \stackrel{\mathsf{def}}{=} [\![ME]\!]_c[\![\langle S, E, \mathsf{ap} : C, D\rangle]\!]_{bc\vec{a}} \qquad (\textsc{Ctr1})$$

$$[\![\langle S, E, \mathsf{ap} : C, D\rangle]\!]_{cb\vec{a}} \stackrel{\mathsf{def}}{=} \nu r\, \overline{b}cr \mid r(d).[\![\langle S, E, C, D\rangle]\!]_{d\vec{a}} \qquad (\textsc{Call})$$

$$[\![\langle S, E, \mathsf{nil}, \langle S', E', C', D'\rangle\rangle]\!]_{b\vec{a}} \stackrel{\mathsf{def}}{=} \nu r\, \overline{r}b \mid r(c).[\![\langle S', E', C', D'\rangle]\!]_{c\vec{a}} \qquad (\textsc{Ret})$$

$$[\![\langle S, E, M, D\rangle]\!]_{\vec{a}} \stackrel{\mathsf{def}}{=} [\![ME]\!]_b[\![\langle S, E, \mathsf{nil}, D\rangle]\!]_{b\vec{a}} \qquad (\textsc{Term})$$

$$[\![\langle \mathrm{Cl} : S, E, C, D\rangle]\!]_{\vec{a}} \stackrel{\mathsf{def}}{=} [\![\mathrm{Cl}]\!]_b[\![\langle S, E, C, D\rangle]\!]_{b\vec{a}} \qquad (\textsc{Stk})$$

$$[\![D]\!]_{cb\vec{a}} \stackrel{\mathsf{def}}{=} [][c/b] \qquad (\textsc{Nil})$$

The encoding proceeds by orderly compiling the λ-terms in the control string and in the stack (rules Ctr2, Ctr1, and Stk). To keep hold of the location of these terms, our encoding asks for a 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 at the head of the control string, we know which function to call with which argument (rule Call).

Notice that rule Ret cannot be generalized to $[\![\langle S, E, \mathsf{nil}, D\rangle]\!]_{b\vec{a}} \stackrel{\mathsf{def}}{=} \nu r\, \overline{r}b \mid r(c).[\![D]\!]_{c\vec{a}}$, for the context $[\![\langle \mathrm{Cl} : S, E, \mathsf{nil}, \mathsf{nil}\rangle]\!]_a$ would reduce, whereas the dump $\langle \mathrm{Cl} : S, E, \mathsf{nil}, \mathsf{nil}\rangle$ does not. Rules Term and Nil allow to show that $[\![\mathsf{term\text{-}to\text{-}dump}(M)]\!]_a = [\![M]\!]_a$.

Example 29. Take for M the term $(\lambda x\mathbf{I}x)V$ where V is a closed value. We have seen in Example 9 that

$$\mathsf{term\text{-}to\text{-}dump}(M) \to^*_{\mathrm{SECD}} \langle\langle V, \emptyset\rangle, \{\langle V, y\rangle\}, \mathsf{nil}, \langle \mathsf{nil}, \{\langle V, x\rangle\}, \mathsf{nil}, \langle \mathsf{nil}, \emptyset, \mathsf{nil}, \mathsf{nil}\rangle\rangle\rangle$$

Then, encoding the thus obtained dump into a pi-context, using rules Stk, Ret, Ret, Nil, we get

$$[\![V]\!]_b[\nu r'\, \overline{r'}b \mid \nu r\, r'(c).(\overline{r}c \mid r(a).[])]$$

On the other hand, a adaptation of example 12.4 yields

$$[\![M]\!]_a \to^2_{\mathsf{d}} \sim [\![V]\!]_b[\nu r'\, \overline{r'}b \mid \nu r\, r'(c).\overline{r}c \mid r(a).[]]$$

Notice that the pi contexts above are strongly barbed congruent (see Proposition 32 below).

8.2 Operational correspondence

The *context machine*, Context, is the triple $\langle \mathsf{context}, \rightarrow_{\mathsf{d}}, \sim_{\Delta} \rangle$. Fix a name a not in variable; the dump-to-context *correspondence* is the function defined as follows.[1]

$$\mathsf{dump\text{-}to\text{-}context}(D) \stackrel{\mathsf{def}}{=} [\![D]\!]_a \qquad \text{if } D \text{ is a value dump}$$

Theorem 30. *The* dump-to-context *correspondence is operational.*

To establish this result we need the exchange lemma 18, the alternative encoding 22, and the following four lemmas.

Proposition 31. $\nu r\, [\![M]\!]_a \overline{r}a \mid r(b).[] \sim_{\Delta} [\![M]\!]_a [\nu r\, \overline{r}a \mid r(b).[]]$.

Proof. Case analysis on M. When M is a value, the left hand side of the equation syntactically equals the right hand side. When M is an application the left hand side is structural congruent to

$$\nu r\, '[\![N]\!]_c [\![L]\!]_d \overline{c}br' \mid \nu r\, r'(a).\overline{r}a \mid r(b).[]$$

and the right hand side is structural congruent to

$$\nu r\, '[\![N]\!]_c [\![L]\!]_d \overline{c}br' \mid \nu r\, r'(a).(\overline{r}a \mid r(b).[])$$

We only have to show that $\nu r\, r'(a).\overline{r}a \mid r(b).[] \sim_{\Delta} r'(a).(\overline{r}a \mid r(b).[])$, which should be easy to establish. □

Proposition 32. *If the variables in the domain of E are not free in the hole, then $[\![NE]\!]_a [\![ME]\!]_b \sim_{\Delta} [\![E]\!][\![N]\!]_a [\![M]\!]_b$.*

Proof. A case analysis on M and N; uses the replication lemma 10.3. Suppose that E is the environment $\{\langle x_i, \mathrm{Cl}_i \rangle \mid i = 1, n\}$. Four cases must be analyzed.

Case M, N are variables. Without loss of generality, suppose M is x_1 and N is x_2. The left-hand-side is $[\![\mathrm{Cl}_1]\!]_a [\![\mathrm{Cl}_2]\!]_b$, whereas the right-hand-side is $[\![\mathrm{Cl}_1]\!]_a [\![\mathrm{Cl}_2]\!]_b [\![\mathrm{Cl}_3]\!]_{x_3} \ldots [\![\mathrm{Cl}_n]\!]_{x_n} \ldots$. To obtain the result, garbage collect environment entries x_3 to x_n since these names are not free in the hole.

Case exactly one of M, N is a variable. As above.

Case M and N are not variables. Use the replication lemma 10.3 for each closure in E. □

Theorem 33. *If $D \rightarrow_{SECD} D'$, then $[\![D]\!]_a \rightarrow_{\mathsf{d}} \sim_{\Delta} [\![D']\!]_a$ or $[\![D]\!]_a \sim_{\Delta} [\![D']\!]_a$.*

Proof. A case analysis on the dumps that are in the transition function; uses the exchange lemma 18, the alternative encoding 22, and propositions 28, 31, 32. The relation that holds between $[\![D]\!]_a$ and $[\![D']\!]_a$ is summarised in the table below, according to the SECD reduction rule used.

[1] For dump-to-context to be a total function, add a rule $[\![D]\!]_{\vec{a}} \stackrel{\mathsf{def}}{=} []$ at the bottom of the list of the rules that compose the mapping $[\![D]\!]_{\vec{a}}$ above.

Rule	Relation
VAR	$\equiv$
ABS	$\equiv$
APP	$\sim_\Delta$
CALL	$\rightarrow_{\mathsf{d}}\sim_\Delta$
RET	$\rightarrow_{\mathsf{d}}$

Case VAR. Three sub-cases must be analyzed. The diagrams below summarise the proof.

$$\langle S, E, x\colon M\colon \mathsf{ap}\colon C, D\rangle \xrightarrow{\rightarrow_{\text{SECD}}} \langle E(x)\colon S, E, M\colon \mathsf{ap}\colon C, D\rangle \quad \text{(CTR2)}$$

$$\begin{array}{ccc} \langle \mathrm{Cl}\colon S, E, x\colon \mathsf{ap}\colon C, D\rangle & \xrightarrow{\rightarrow_{\text{SECD}}} & \langle E(x)\colon \mathrm{Cl}\colon S, E, \mathsf{ap}\colon C, D\rangle \\ \downarrow \text{STK;CTR1} & \swarrow \text{STK;STK} & \\ [\![E(x)]\!]_b[\![\mathrm{Cl}]\!]_c[\![\langle S, E, \mathsf{ap}\colon C, D\rangle]\!]_{bca} & & \end{array}$$

$$\langle S, E, x, D\rangle \xrightarrow{\rightarrow_{\text{SECD}}} \langle E(x)\colon S, E, \mathsf{nil}, D\rangle \quad \text{(CTR2)}$$

Case ABS. Similar.

Case APP. As above, three sub-cases must be analyzed.

$$\begin{array}{ccc} \langle S, E, MN\colon L\colon \mathsf{ap}\colon C, D\rangle & \xrightarrow{\rightarrow_{\text{SECD}}} & \langle S, E, N\colon M\colon \mathsf{ap}\colon C, D\rangle \\ \downarrow \text{CTR2} & & \downarrow \text{CTR2;CTR1;CALL} \\ [\![E]\!][\![M]\!]_b[\![N]\!]_cB & \xrightarrow{\sim_\Delta} & [\![ME]\!]_b[\![NE]\!]_cB \end{array}$$

where the context B is $\nu r\overline{b}cr \mid r(d).[\![\langle S, E, L\colon \mathsf{ap}\colon C, D\rangle]\!]_{da}$. The $\sim_\Delta$ step follows from proposition 32.

$$\begin{array}{ccc} \langle \mathrm{Cl}\colon S, E, MN\colon \mathsf{ap}\colon C, D\rangle & \xrightarrow{\rightarrow_{\text{SECD}}} & \langle \mathrm{Cl}\colon S, E, N\colon M\colon \mathsf{ap}\colon \mathsf{ap}\colon C, D\rangle \\ \downarrow \text{STK;CTR1;CALL} & & \downarrow \text{CTR2;CTR1;CALL;STK;CALL} \\ [\![\mathrm{Cl}]\!]_e[\![(MN)E]\!]_dB & \xrightarrow{\sim_\Delta} & [\![NE]\!]_b[\![ME]\!]_c[\nu r\,\overline{c}br \mid r(d).[\![\mathrm{Cl}]\!]_eB] \end{array}$$

where the context B is $\nu r\,'\overline{e}dr' \mid r'(f).[\![\langle S, E, C, D\rangle]\!]_{fa}$. The $\sim_\Delta$ step follows from proposition 32, the alternative encoding 22, and the exchange lemma 18.

$$\begin{array}{ccc} \langle S, E, MN, D\rangle & \xrightarrow{\rightarrow_{\text{SECD}}} & \langle S, E, N\colon M\colon \mathsf{ap}, D\rangle \\ \downarrow \text{TERM;STK} & & \downarrow \text{CTR1;CTR2;CALL} \\ [\![ME]\!]_dB & \xrightarrow{\sim_\Delta} & [\![NE]\!]_b[\![ME]\!]_c[\nu r\,\overline{c}br \mid r(d).B] \end{array}$$

where B is the context $[\![\langle S, E, \mathsf{nil}, D\rangle]\!]_{da}$. The $\sim_\Delta$ step follows from proposition 32, the alternative encoding 22.

Case CALL.

$$\begin{array}{ccc}
\langle\langle \lambda x M, E'\rangle \colon \mathrm{Cl} \colon S, E, \mathsf{ap} \colon C, D\rangle & \xrightarrow{\rightarrow \mathrm{SECD}} & \langle \mathsf{nil}, E'\{\mathrm{Cl}/x\}, M, \langle S, E, C, D\rangle\rangle \\
\Big\downarrow \textsc{Stk};\textsc{Stk};\textsc{Call} & & \Big\downarrow \textsc{Term};\textsc{Ret} \\
[\![\mathrm{Cl}]\!]_c [\![E']\!] [\![\lambda x M]\!]_b [\nu r\, \overline{b} c r \mid r(d).B] & \xrightarrow{\rightarrow_{\mathsf{d}} \sim_{\Delta}} & [\![E'\{\mathrm{Cl}/x\}]\!] [\![M]\!]_b [\nu r\, \overline{r} b \mid r(d).B]
\end{array}$$

where B is the context $[\![\langle S, E, C, D\rangle]\!]_{da}$. For the reduction step $\rightarrow_{\mathsf{d}}$, notice that only values are present in closures so that the process $\nu r\, \overline{b} c r \mid r(d).B$ is under no prefix (proposition 28). The $\sim_{\Delta}$ step follows from the garbage collection of the environment entry at b, and from proposition 31.

Case RET.

$$\begin{array}{ccc}
\langle \mathrm{Cl} \colon S, E, \mathsf{nil}, \langle S', E', C', D'\rangle\rangle & \xrightarrow{\rightarrow \mathrm{SECD}} & \langle \mathrm{Cl} \colon S', E', C', D'\rangle \\
\Big\downarrow \textsc{Stk};\textsc{Ret} & & \Big\downarrow \textsc{Stk} \\
[\![\mathrm{Cl}]\!]_b [\nu r\, \overline{r} b \mid r(c).[\![\langle S', E', C', D'\rangle]\!]_{ca}] & \xrightarrow{\rightarrow_{\mathsf{d}}} & [\![\mathrm{Cl}]\!]_b [\![\langle S', E', C', D'\rangle]\!]_{ba}
\end{array}$$

Once again notice that, since only values are present in closures, the process $\nu r\, \overline{r} b \mid r(c).B$ is under no prefix (proposition 28). □

Theorem 34. *If $D \uparrow$, then $[\![D']\!]_a \uparrow$.*

Proof. Examining the table in the proof of Theorem 33, one needs to show that when D diverges there there are infinitely many transitions by rules CALL or RET. In other words, in any reduction, the number of consecutive applications of rules VAR, ABS, and APP is finite. That the above holds becomes obvious by inspection of the transitions rules of the SECD machine, in Section 4. There can only be a finite number of consecutive applications of rules VAR, ABS, and APP since the control string has a finite number of elements, and each of its elements (ap or a lambda term) is finite. □

Proof of Theorem 30. We show that the three diagrams in the definition of the operational correspondence hold. The first corresponds to Theorem 33; the second to the contrapositive of Theorem 34.

For the third diagram, we show its contrapositive, that is, if $D \not\rightarrow_{\mathrm{SECD}}$ then $[\![D]\!]_a \not\rightarrow$. We proceed by analyzing the dumps that do not reduce. They are of the form $\langle\langle E, M\rangle \colon \_, \_, \mathsf{ap} \colon \_, \_\rangle$ for M not an application, $\langle \mathrm{Cl}, \_, \mathsf{ap} \colon \_, \_\rangle$, $\langle \mathrm{Cl} \colon \_, \_, \mathsf{nil}, \mathsf{nil}\rangle$, $\langle \mathsf{nil}, \_, \mathsf{nil}, \_\rangle$, and nil. Clearly their encodings do not reduce. □

Chapter 9

Conclusion

We present an encoding for the call-by-value λ-calculus into the π-calculus that is barbed-congruent to Milner's well-known "light" encoding, and saves one π-reduction-step per β-reduction-step.

Based on the new lambda-to-pi encoding, we describe an encoding of the SECD machine into the π-calculus, and prove that is constitutes an operational correspondence.

The impact of the call by value-λ to π encoding here proposed on actual, π-based, programming languages [13, 20] should be analyzed. Also, further optimizations for particular patterns of λ-terms could be pursued.

The encoding of the SECD machine opens perspectives of encoding others machines, thus providing for the study of other lambda reduction strategies in the π-calculus.

Sangiorgi presents Milner's encoding using a continuation-passing scheme [16]. It should be interesting to investigate whether there is a continuation-passing-style transform that would yield the encoding here presented.

The local π-calculus [7] seems a promising setting to study the encoding, obviating the need for typed contexts. All the proofs in the paper ultimately rely on a handful of replication lemmas. It remains to see if these theorems are valid in the local π-calculus.

Acknowledgements. Special thanks to D. Sangiorgi and A. Ravara for fruitful discussions. The Wednesday Morning Club at the University of Lisbon provided important feedback, N. Barreiro in particular. This work is partially supported by project PRAXIS/P/EEI/120598/98 DiCoMo.

Bibliography

[1] 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.

[2] Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[3] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *5th European Conference on Object-Oriented Programming*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.

[4] Naoki Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings of LICS '97*, pages 128–139. Computer Society Press, July 1997.

[5] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4), 1964. Referred in [14].

[6] L. Lopes, F. Silva, and V. Vasconcelos. A virtual machine for the TyCO process calculus. In *PPDP'99*, volume 1702 of *LNCS*, pages 244–260. Springer-Verlag, September 1999.

[7] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of ICALP '98*, volume 1443 of *LNCS*, pages 856–867. Springer, July 1998.

[8] Robin Milner. Functions as processes. Rapport de Recherche RR-1154, INRIA Sophia-Antipolis, 1990. Final version in [9].

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

[10] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100:1–77, 1992.

[11] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proceedings of ICALP '92*, volume 623 of *LNCS*, pages 685–695. Springer, 1992.

[12] 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.

[13] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. CSCI Technical Report 476, Indiana University, March 1997.

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

[15] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[16] Davide Sangiorgi. Interpreting functions as π-calculus processes: a tutorial. Unpublished, August 1998.

[17] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, LFCS, University of Edinburgh, June 1996. CST-126-96 (also published as ECS-LFCS-96-345).

[18] 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.

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

[20] Vasco T. Vasconcelos and Rui Bastos. Core-TyCO, the language definition, version 0.1. DI/FCUL TR 98–3, Department of Computer Science, University of Lisbon, March 1998.