Self-applicable partial evaluation for the pi-calculus

Marc Gengler Matthieu Martel

Ecole Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme (LIP)
46, Allée d'Italie
69364 Lyon Cedex 07, France
E-mail: [Marc.Gengler,Matthieu.Martel]@lip.ens-lyon.fr

Abstract

In this paper, we are interested in self-applicable partial evaluation for the pi-calculus, a language which models the concurrent behavior of communicating processes. We use the classic three-steps methodology. First, we write a meta-interpreter for the language. Second, we introduce an abstract analysis that determines which operations (communications) can be executed at compile-time. The notion of well-annotatedness of terms is defined. Finally, we exhibit the self-applicable partial evaluator which is applied to well-annotated terms, and we prove its correctness with respect to the interpreter. This approach is compatible with Futamura's projections. Proofs of correctness are based on the notion of weak reduction equivalence.

Keywords: Partial Evaluation, Parallelism, Pi-Calculus, Meta-Interpretation, Binding-Time Analysis.

1 Introduction

Partial evaluation is an optimization technique which consists of specializing programs with respect to the known part of their inputs. Considerable work has been done in this area [1], however, most of the research has focused on sequential functional or imperative languages. In this paper, we are interested in partial evaluation for the π-calculus [9]. This language, *à la* CCS [8], describes the behavior of concurrent processes which interact using explicit communications over channels. π-calculus is well-suited to model the principles of concurrent as well as object-oriented languages [16].

Even though one can expect great improvements from partial evaluation techniques and related abstract interpretation methods in those areas, only few attempts have been made. First, concerning parallelism, applications could include load balancing, communication optimization [4], and run-time parallelization using techniques similar to these used in [5] for deferred compilation. Nevertheless, only some attempts have been done. For instance, [3] presents an on-line partial evaluator for a CCS-like language. Second, concerning object-oriented languages, reflection is an important topic and partial evaluation might address the lack of efficiency due to high-level abstractions [7].

Our goals are first to show that partial evaluation for concurrent languages is possible, and then to formalize it. Our approach uses the well established three-steps methodology developed in [2, 10, 11, 12, 17] for partial evaluation of the λ-calculus, in the context of concurrent languages.

(i) We define a meta-interpreter for the π-calculus. We prove its correctness w.r.t. to the notion of *weak reduction equivalence* [9], denoted $\dot{\approx}$. $\dot{\approx}$ is defined in Section 2. A function $\lceil . \rceil$ describes how programs can be encoded in the π-calculus. We also define the interpreter, denoted `Eval`. The correctness criterion states that applying `Eval` to the encoding $\lceil P \rceil$ of a π-term P yields to a program $\dot{\approx}$-equivalent to P. Reflection of `Eval` is crucial to preserve properties of partial evaluation such as Futamura's projections [17]. However, as discussed in [14], it interests for its own, since reflection is a property one usually tries to establish when studying a new formal language such as π-calculus.

(ii) We propose a *binding-time analysis* (BTA for short) for the language, that determines which communications can be executed at compile-time. This BTA annotates terms, and outputs two-level terms [2] in which dynamic operations are underlined. As far as we are aware, this is the first binding-time analysis proposed for the π-calculus. We also introduce the notion of well-annotatedness, needed to define the correctness of partial evaluation.

(iii) Finally, we achieve our primary objective by introducing a self-applicable partial evaluator `Pev`. `Pev` is applied to two-level terms, encoded using a function $\lfloor . \rfloor$. Again, the proofs are based on the notion of weak reduction equivalence. We state that applying `Pev` to well-annotated terms preserves the $\dot{\approx}$-equivalences.

In Section 2, we formalize the concepts of partial evaluation and introduce notations and definitions concerning the π-calculus. Section 3 presents the mechanisms used in the remainder of the paper, using a more familiar language, the λ-calculus. Sections 4, 5 and 6 respectively deal with the interpreter, the binding-time analysis, and the partial evaluator. Results are discussed in Section 7.

2 Preliminary Definitions

Following the presentation of [17], a *programming language* $\mathcal{L}$ is specified by a set $\mathcal{P}$ of $\mathcal{L}$-programs, a set Δ of $\mathcal{L}$-data, and a family $\mathcal{S} = (\mathcal{S}_n)_{n\in\mathbb{N}}$, of $(n+2)$-ary relations ($\mathcal{S}$ is the semantics of $\mathcal{L}$). Let $\rho \in \mathcal{P}$, and $\delta_1, \ldots, \delta_n, \delta \in \Delta$, $(\rho, \delta_1, \ldots, \delta_n, \delta) \in \mathcal{S}_n$ if and only if δ is the result of the application of the program ρ to the data $\delta_1 \ldots \delta_n$.

An *interpreter* is a program which uses two kinds of data: the program ρ to be executed, and the data $\delta_1 \ldots \delta_n$ to be applied to ρ. Since a self-interpreter for the language $\mathcal{L}$ is a program $\texttt{Eval} \in \mathcal{P}$, ρ must be encoded under the form of $\mathcal{L}$-data, in order to be understood by `Eval`.

Definitions 1 (Encoding and meta-interpretation) *A self-interpreter related to an encoding function* $\lceil.\rceil : \mathcal{P} \to \Delta$ *is a program* $\texttt{Eval} \in \mathcal{P}$ *such that*

$$\begin{array}{c}(\texttt{Eval}, \lceil\rho\rceil, \delta_1, \ldots, \delta_n, \delta) \in \mathcal{S}_{n+1} \\ \Longleftrightarrow \\ (\rho, \delta_1, \ldots, \delta_n, \delta) \in \mathcal{S}_n\end{array} \quad (1)$$

Equation (1) states that the interpretation of the encoding of a program ρ produces the result δ specified by the semantics of ρ.

A *partial evaluator* is a program `Pev` which specializes a program ρ with respect to the known part (the *static* part) of its data. Note that it is always possible to concatenate all the static [resp. dynamic] data used by ρ. Thus, we can assume that any program ρ only uses two data δ_1 and δ_2, which are respectively the concatenations of all the static and dynamic ones. In order to determine the static parts of a program ρ, a *binding-time analysis* $\Phi : \mathcal{P} \to \mathcal{P}_a$ is used. It yields an annotated version $\Phi(\rho) \in \mathcal{P}_a$ of ρ. These annotated terms must again be encoded as $\mathcal{L}$-data, in order to be understood by the partial evaluator.

Definitions 2 (Analysis and Partial evaluation) *A self-applicable partial evaluator related to an analysis function* Φ *and to an encoding function* $\lfloor.\rfloor : \mathcal{P}_a \to \Delta$ *is a program* $\texttt{Pev} \in \mathcal{P}$, $\texttt{Pev} : \Delta \to \Delta$ *such that*

$$(\texttt{Pev}, \lfloor\Phi(\rho)\rfloor, \delta_1, \rho') \in \mathcal{S}_2 \Rightarrow (\texttt{Eval}, \rho', \delta_2, \delta) \in \mathcal{S}_2 \quad (2)$$

Equation (2) states that partially evaluating a program ρ w.r.t. the part δ_1 of its data leads to a program ρ' such that $(\rho'\ \delta_2) = (\rho\ \delta_1\ \delta_2)$. Note that using (1), Equation (2) is equivalent to

$$(\texttt{Pev}, \lfloor\Phi(\rho)\rfloor, \delta_1, \rho') \in \mathcal{S}_2 \Rightarrow (\rho, \delta_1, \delta_2, \delta) \in \mathcal{S}_2$$

In this paper, we are interested in the case $\mathcal{L} = \pi$, namely, in self-interpretation and self-applicable partial evaluation for (asynchronous) π-*calculus* [9]. Programs, denoted by π-terms, are generated by the grammar:

$$\left\{\begin{array}{lcl} P & ::= & \sum_{i=1}^{n} \pi_i.P_i \mid P\|Q \mid !P \mid (\nu x)P \\ \pi & ::= & \overline{\alpha}[x] \mid \alpha(x)\end{array}\right.$$

Channels (also called *names*) are the only values in the language. They are used to indicate the names of communication channels as well as transmitted values. Sending (resp. receiving) a message x over a channel α is done by the use of the instruction $\overline{\alpha}[x]$ (resp. $\alpha(x)$). The name α used in such a communication π is called the *subject* of π. $P\|Q$ represents the concurrent execution of P and Q. $!P$ represents the replication of P, i.e. $!P$ is $P\|P\|P\|\ldots$ as many times as needed. (νx) creates a new channel x for communications, and $\sum_{i=1}^{n} \pi_i.P_i$ represents the non-deterministic choice among processes. Let $\to$ be the *reduction relation* over terms. The semantics of the sum operator is:

$$\begin{array}{c}\Big(\ldots + \ldots + \overline{\alpha}[x].P\Big) \parallel \Big(\ldots + \ldots + \alpha(y).Q\Big) \\ \to P \parallel Q\{y \leftarrow x\}\end{array} \quad (3)$$

Remark that each term in a non-deterministic sum must start with a communication. The other reduction rules are:

$$\frac{P \to P'}{P\|Q \to P'\|Q} \qquad \frac{Q \to Q'}{P\|Q \to P\|Q'} \qquad \frac{P \to P'}{(\nu x)P \to (\nu x)P'}$$

The reflective transitive closure of $\to$ is denoted $\to^*$. The following abbreviations are also used. $\alpha(x_1 \ldots x_n)$ denotes $\alpha(x_1)\ldots\alpha(x_n)$, $\overline{\alpha}[x_1 \ldots x_n]$ denotes $\overline{\alpha}[x_1]\ldots\overline{\alpha}[x_n]$ and $(\nu x_1 \ldots x_n)$ denotes $(\nu x_1)\ldots(\nu x_n)$. $\mathbf{0}$ denotes the null process, i.e. the empty sum of non-deterministic operations. α and $\overline{\alpha}$ abbreviate $\alpha()$ and $\overline{\alpha}[]$, i.e. the communications of empty messages, also called signals.

We introduce now two standard equivalence relations between processes. First, the *structural congruence relation*, denoted $\equiv$, identifies the terms whose syntax differs, but that express the same meaning. Structurally equivalent terms act identically in any context, and substituting one for the other does not modify the behavior of the system.

Definition 3 (Structural congruence) *The structural congruence* $\equiv$ *is the smallest congruence relation over terms such that*

- *(i)* $\equiv$ *contains the alpha-equivalence relation.*
- *(ii)* $\|$ *and* $+$ *are associative, symmetric and* $\mathbf{0}$ *is the neutral element for these operators.*
- *(iii)* $!P \equiv P\|!P$, $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$.
- *(iv) If* P *does not contain any free occurrence of* x*, then* $(\nu x)(P\|Q) \equiv P\|(\nu x)Q$.

The reduction relation $\to$ is extended by the rule:

$$\frac{P \equiv Q \quad Q \to Q' \quad Q' \equiv P'}{P \to P'}$$

Also note that the following property used in the proofs in Sections 4 and 6 is easily verified:

$$x \text{ not free in P} \Rightarrow (\nu x)P \equiv P \quad (4)$$

The second relation, denoted $\dot{\approx}$, is the *weak reduction equivalence relation*. It identifies terms which look equal from an external point of view. If $P \dot{\approx} Q$, then any process R cannot distinguish between P and Q by observing its interactions. The notions of unguardness and observability are fundamental to formalize $\dot{\approx}$.

Definitions 4 (Unguardness and observability) *Let P and Q be two processes. Q occurs* unguarded *in P if Q has some occurrence in P which is not under a prefix π. P is* observable *at α, written $P \downarrow_\alpha$, if $\pi.Q$ occurs unguarded in P for some Q and π such that α is the subject of π.*

For instance, $P \equiv (\alpha(x).Q_1 \| Q_2) \downarrow_\alpha$, since P can immediately communicate on α. Similarly, $((\nu x)\alpha(y).P) \downarrow_\alpha$ or $((\nu x)\alpha(x).P) \downarrow_\alpha$. However, $(\alpha(x).\beta(y)\ .P) \not\downarrow_\beta$ because no communication on β is possible before $\alpha(x)$ is reduced (the communication on β is guarded by the one on α). Note also that $P \equiv ((\nu x)x(y).P) \not\downarrow_x$, even if $x(y)$ is unguarded, because x is a new channel unknown outside of P.

Using these notions, $\dot{\approx}$ is the relation identifying terms observable on the same channels. This property must be preserved under reduction.

Definition 5 (Weak reduction equivalence) *The weak reduction equivalence $\dot{\approx}$ is the largest equivalence relation $\mathcal{R}$ over processes such that $\mathcal{R}(P,Q)$ implies :*

$$\text{(i)}\ (P \to P') \Rightarrow \Big(\exists Q'\ :\ (Q \to^* Q' \text{ and } \mathcal{R}(P',Q'))\Big).$$

$$\text{(ii)}\ (\forall \alpha,\ \Big(P \downarrow_\alpha) \Rightarrow \exists Q'\ :\ (Q \to^* Q' \text{ and } Q' \downarrow_\alpha)\Big).$$

Let us remark that, since $\dot{\approx}$ is symmetric, $P \dot{\approx} Q$ also implies

$$(Q \to Q') \Rightarrow \Big(\exists P'\ :\ (P \to^* P' \text{ and } P' \equiv Q')\Big)$$

and

$$(\forall \alpha,\ \Big(Q \downarrow_\alpha) \Rightarrow \exists P'\ :\ (P \to^* P' \text{ and } P' \downarrow_\alpha)\Big)$$

3 Related work

This section briefly describes partial evaluation for pure λ-calculus [10, 11, 12, 17]. It introduces the methodology of self-applicable partial evaluation of a formal language. This scheme, considered here for the well-known λ-calculus, will be used in Sections 4, 5 and 6 for self-applicable partial evaluation of the π-calculus.

As indicated in Section 2, three steps have to be considered. The first one concerns the meta-interpreter. An encoding of λ-terms is given, and the meta-interpreter which executes the encoded terms is introduced. The second step concerns the analysis of programs. It yields annotated terms which also have to be encoded in the λ-calculus. The last step concerns the partial evaluator which treats annotated terms.

Programs in the λ-calculus have to be encoded as data. The representation scheme $\lceil . \rceil_\Lambda$ is given in Figure 1. It combines the notions of signature representation, and of higher order abstract syntax [13]. Variables a, b and c act as switch operators. They indicate the syntactic category encoded by $\lceil t \rceil_\Lambda$. A a indicates that t is a variable. b and c respectively describe application and abstraction.

The evaluator $\mathtt{Eval}_\Lambda$, also given in Figure 1, executes λ-terms encoded with the function $\lceil . \rceil_\Lambda$. The operation corresponding to one of the indicators a, b or c is applied to the variable x, the terms $\lceil e_1 \rceil_\Lambda$ and $\lceil e_2 \rceil_\Lambda$ or the term $(\lambda x.\lceil e \rceil_\Lambda)$. $\mathtt{Y}$ denotes the fix-point operator.

$$\begin{array}{rcl} \lceil x \rceil_\Lambda & \equiv & \lambda abc.a\ x \\ \lceil e_1\ e_2 \rceil_\Lambda & \equiv & \lambda abc.b\ \lceil e_1 \rceil_\Lambda\ \lceil e_2 \rceil_\Lambda \\ \lceil \lambda x.e \rceil_\Lambda & \equiv & \lambda abc.c\ (\lambda x.\lceil e \rceil_\Lambda) \end{array}$$

$$\begin{array}{rl} \mathtt{Eval}_\Lambda \equiv \mathtt{Y}\ (\lambda e.\lambda m.m & (\lambda x.x) \\ & (\lambda mn.(e\ m)(e\ n)) \\ & (\lambda m.\lambda v.e\ (m\ v))) \end{array}$$

$$\mathtt{Y} \equiv \lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x))$$

Figure 1: Encoding and evaluation of λ-terms.

The correctness property consists of proving that $\mathtt{Eval}_\Lambda$ applied to a program $\lceil t \rceil_\Lambda$ yields a term semantically equivalent to t. In λ-calculus, this is done using the β-equivalence relation $=_\beta$.

$$\mathtt{Eval}_\Lambda\ \lceil t \rceil_\Lambda\ =_\beta\ t \tag{5}$$

The next step is to annotate programs. Annotations consist of underlining the dynamic operations. This is formalized by the notion of *two-level λ-terms.*

Definition 6 (Two-level λ-terms) *A two-level λ-term is a term generated by the grammar*

$$t ::= x \mid (t_1\ t_2) \mid \lambda x.t \mid (t_1 \underline{\ } t_2) \mid \underline{\lambda} x.t$$

x, $(t_1\ t_2)$, and $\lambda x.t$ represent static terms which can be reduced by the partial evaluator $\mathtt{Pev}_\Lambda$. $(t_1 \underline{\ } t_2)$ and $\underline{\lambda} x.t$ represent dynamic terms. In this case, $\mathtt{Pev}_\Lambda$ partially evaluates the sub-expressions and outputs the encoding $\lceil r \rceil_\Lambda$ of the residual program r. Binding-time analyses producing two-level λ-terms can be found in [11, 12, 17].

$$\begin{array}{rcl} \lfloor x \rfloor_\Lambda & \equiv & \lambda abcde.a\ x \\ \lfloor e_1\ e_2 \rfloor_\Lambda & \equiv & \lambda abcde.b\ \lfloor e_1 \rfloor_\Lambda\ \lfloor e_2 \rfloor_\Lambda \\ \lfloor \lambda x.e \rfloor_\Lambda & \equiv & \lambda abcde.c\ (\lambda x.\lfloor e \rfloor_\Lambda) \\ \lfloor e_1 \underline{\ } e_2 \rfloor_\Lambda & \equiv & \lambda abcde.d\ \lfloor e_1 \rfloor_\Lambda\ \lfloor e_2 \rfloor_\Lambda \\ \lfloor \underline{\lambda} x.e \rfloor_\Lambda & \equiv & \lambda abcde.e\ (\lambda x.\lfloor e \rfloor_\Lambda) \end{array}$$

$$\begin{array}{rl} \mathtt{Pev}_\Lambda \equiv \mathtt{Y}\ (\lambda p.\lambda m.m & (\lambda x.x) \\ & (\lambda mn.(p\ m)(p\ n)) \\ & (\lambda m.\lambda v.p\ (m\ v)) \\ & (\lambda mn.\lambda abc.b\ (p\ m)(p\ n)) \\ & (\lambda m.\lambda abc.c\ (\lambda v.p\ (m\ (\lambda abc.a\ v)))) \end{array}$$

Figure 2: Encoding and partial evaluation of two-level λ-terms.

Finally, $\mathtt{Pev}_\Lambda$ is introduced. Annotated terms are encoded using the function $\lfloor . \rfloor_\Lambda$ given in Figure 2. $\lfloor . \rfloor_\Lambda$ extends $\lceil . \rceil_\Lambda$ in order to encode the dynamic operations. Here, five variables are used. d and e respectively represent dynamic application and abstraction. Figure 2 also describes $\texttt{Pev}_\Lambda$, the partial evaluator for the λ-calculus which interprets two-level λ-terms. Static terms are reduced, and when a dynamic operation is encountered, its sub-expressions are partially evaluated and the resulting program is encoded as in $\lceil . \rceil_\Lambda$.

Proposition 7 (Correctness of $\texttt{Pev}_\Lambda$) *Let t be a λ-term with arity n and t_a an annotated version of t. A partial evaluator $\texttt{Pev}_\Lambda$ is correct w.r.t. t_a's annotation system if*

$$\forall k \leq n,\ \forall s_1 \dots s_k,\ \texttt{Pev}_\Lambda\ \lfloor t_a \rfloor_\Lambda\ s_1 \dots s_k\ =_\beta\ \lceil t\ s_1 \dots s_k \rceil_\Lambda$$

This means that, $(\texttt{Pev}_\Lambda\ \lfloor t_a \rfloor_\Lambda)$ applied to an acceptable freely chosen number of arguments will behave like the program which encodes the application of t to the same arguments. More details about the correctness of the partial evaluator described in Figure 2 can be found in [12] and [17].

4 The interpreter

In this section, we introduce a self-interpreter, denoted `Eval`, for π-calculus. First, we describe the encoding function $\lceil . \rceil$ of the terms. Next, we introduce `Eval` and prove its correctness. Since the π-calculus is a communication based language, program sources are processes which send messages to the interpreter. `Eval` receives and executes the encoded operations.

To keep the evaluator readable we assume without lost of generality the following fact. Every non-deterministic sum of processes $\sum_{i=1}^{n} \pi_i.P_i$ has a constant size n and can be rewritten

$$\sum_{i=1}^{\frac{n}{2}} \alpha_i(x_i).P_i + \sum_{i=\frac{n}{2}+1}^{n} \overline{\alpha_i}[x_i].P_i$$

This hypothesis only permits to obtain a more compact representation of `Eval`. It can be removed either by using reserved channels indicating the length of the sums or by encoding integers, which can easily be achieved in π-calculus.

As indicated in Section 2, programs in the π-calculus have to be encoded as data. Since the only entity in the language is the process, all those data are represented under that form. Our encoding is close to the one described in Section 3 for pure λ-calculus. Letters from a to e denote reserved channels. A process P is encoded onto a channel γ_0 by the function $\lceil . \rceil$ described in Figure 3.

Names $\overline{a}$ to $\overline{e}$ indicate which kind of process is encoded. A $\overline{a}$ encodes the null process $\mathbf{0}$. $\overline{b}$ is used to indicate a parallel composition. In this case, two new channels γ_1 and γ_2 are created and sent onto γ_0. Next, the processes P_1 and P_2 are encoded recursively, respectively onto γ_1 and γ_2. Name $\overline{c}$ is sent to indicate the occurrence of the replication operator. In this case, the following action is done repeatedly. A new channel γ is created, and sent onto γ_0. It is followed by the encoding onto γ of the process to replicate. $\lceil (\nu x)P, \gamma_0 \rceil$ produces $\gamma_1(x).\lceil P, \gamma_2 \rceil$. The message $\overline{d}$ indicates the beginning of such an encoding. Channels γ_1 and γ_2 are created and sent onto γ_0. The interpreter will send the new name on γ_1 and $\lceil P, \gamma_2 \rceil$ is recursively encoded. Finally, in order to encode the sum of n processes, the channels $\gamma_1 \dots \gamma_n$ and $\delta_1 \dots \delta_n$ are created and sent onto γ_0. The prefixes of the P_i's are also sent onto γ_0. The encoding of a reception reads on a channel δ the new value, and the process continuation encoding is recursively computed. Concerning the emission, δ is only used as a signal since no value is needed to encode the continuation.

The interpreter `Eval` which evaluates programs encoded on the channel γ_0 is given Figure 3.

`Eval` first reads the names $a \dots e$ of the case indicators on γ_0. Next, it receives the description of the operation to execute, under the form of a name from a to e. A a indicates the end of the execution. When b is encountered, the interpreter reads on γ_0 the locations γ_1 and γ_2 of the two continuations and executes $(\texttt{Eval}\ \gamma_1)$ concurrently with $(\texttt{Eval}\ \gamma_2)$. Note that `Eval` is duplicated, and $(\texttt{Eval}\ \gamma_1)$ [resp. $(\texttt{Eval}\ \gamma_2)$] evaluates $\lceil P_1, \gamma_1 \rceil$ [resp. $\lceil P_2, \gamma_2 \rceil$]. c indicates that the process has to be replicated. Locations γ of the continuations are read on γ_0 and $(\texttt{Eval}\ \gamma)$ is done as many times as required. d precedes a restriction. Names γ_1 and γ_2 are read on γ_0. The new name x is created and sent to the encoding through γ_1. Next, $(\texttt{Eval}\ \gamma_2)$ is executed. An occurrence of e is followed by reading on γ_0 the continuation channels $\gamma_1, \dots, \gamma_n$, the names $\delta_1, \dots \delta_n$, and the prefixes of the P_i's. When a value v is received on α_i, $1 \leq i \leq \frac{n}{2}$, it is sent on δ_i to the encoding. The substitution is done in $\lceil P_i, \gamma_i \rceil$, and the resulting term is

$$\lceil P_i\{x_i \leftarrow v\}, \gamma_i \rceil \parallel (\texttt{Eval}\ \gamma_i) \tag{6}$$

Concerning emissions, no value has to be transmitted. Communications on δ_i, $\frac{n}{2} + 1 \leq i \leq n$, are simple signals.

Finally, note that `Eval` uses recursive calls. These calls are not allowed in pure π-calculus. However recursion may easily be simulated in the language as described in [9]. Let ψ be a reserved channel used for recursion. The principle consists of substituting, for all γ, the term $\overline{\psi}[\gamma].\texttt{Eval}$ to any occurrence of $(\texttt{Eval}\ \gamma)$ in `Eval`. Intuitively, communications are done instead of recursive calls. Additionally, $!\psi(\gamma_0).\texttt{Eval}$ is substituted to $\texttt{Eval}(\gamma_0)$. This terms creates a new evaluator for each recursive call. The argument is read on ψ.

In order to prove the correctness of the interpreter, we introduce the following notation:

$$\mathbf{E}(P) \equiv (\nu\gamma_0)\Big(\lceil P, \gamma_0 \rceil \parallel (\texttt{Eval}\ \gamma_0)\Big) \tag{7}$$

This means that $\mathbf{E}(P)$ denotes the application of $(\texttt{Eval}\ \gamma_0)$ to $\lceil P, \gamma_0 \rceil$. The encoded term transmits the source program to the evaluator, via the restricted channel γ_0. Proofs are based on the notion of weak reduction equivalence. The choice of this relation is motivated by the fact that, as mentioned earlier, terms equivalent by weak reduction are not distinguishable from an external point of view.

Lemma 8 (Compositionality of E) *The following properties hold.*

(i) $\mathbf{E}(P_1 \| P_2) \dot{\approx} \mathbf{E}(P_1) \| \mathbf{E}(P_2)$.

(ii) $\mathbf{E}(!P) \dot{\approx}\ !\mathbf{E}(P)$.

(iii) $\mathbf{E}((\nu x)P) \dot{\approx} (\nu x)\mathbf{E}(P)$.

(iv) $\mathbf{E}(\sum_{i=1}^{n} \pi_i.P_i) \dot{\approx} \sum_{i=1}^{n} \pi_i.\mathbf{E}(P_i)$.

PROOF

For each case, for all α, both terms of the equivalence are unobservable at α before reduction, since they communicate

$$
\begin{aligned}
\lceil \mathbf{0}, \gamma_0 \rceil &= (\nu abcde)\overline{\gamma_0}[abcde].\overline{a} \\
\lceil P_1 \| P_2, \gamma_0 \rceil &= (\nu abcde)\overline{\gamma_0}[abcde].\overline{b}.(\nu\gamma_1\gamma_2)\overline{\gamma_0}[\gamma_1\gamma_2].(\lceil P_1, \gamma_1\rceil \| \lceil P_2, \gamma_2 \rceil) \\
\lceil !P, \gamma_0 \rceil &= (\nu abcde)\overline{\gamma_0}[abcde].\overline{c}.!((\nu\gamma)\overline{\gamma_0}[\gamma].\lceil P, \gamma\rceil) \\
\lceil (\nu x)P, \gamma_0 \rceil &= (\nu abcde)\overline{\gamma_0}[abcde].\overline{d}.(\nu\gamma_1\gamma_2)\overline{\gamma_0}[\gamma_1\gamma_2].\gamma_1(x).\lceil P, \gamma_2 \rceil \\
\lceil \sum_{i=1}^{n} \pi_i.P_i, \gamma_0 \rceil &= \begin{array}{l} (\nu abcde)\overline{\gamma_0}[abcde].\overline{e}. \\ \quad (\nu\gamma_1\ldots\gamma_n\delta_1\ldots\delta_n)\overline{\gamma_0}[\gamma_1\ldots\gamma_n\delta_1\ldots\delta_n\alpha_1\ldots\alpha_n x_1\ldots x_n]. \\ \quad \left(\sum_{i=1}^{\frac{n}{2}} \delta_i(x_i).\lceil P_i, \gamma_i\rceil + \sum_{i=\frac{n}{2}+1}^{n} \delta_i.\lceil P_i, \gamma_i \rceil\right) \end{array}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{Eval}(\gamma_0) \equiv\ & \gamma_0(abcde). \\
& (\ \ a.\mathbf{0} \\
& + b.\gamma_0(\gamma_1\gamma_2).((\mathtt{Eval}\ \gamma_1) \parallel (\mathtt{Eval}\ \gamma_2)) \\
& + c.!(\gamma_0(\gamma).(\mathtt{Eval}\ \gamma)) \\
& + d.\gamma_0(\gamma_1\gamma_2).(\nu x)\overline{\gamma_1}[x].(\mathtt{Eval}\ \gamma_2) \\
& + e.\gamma_0(\gamma_1\ldots\gamma_n\delta_1\ldots\delta_n\alpha_1\ldots\alpha_n x_1\ldots x_n). \\
& \qquad \left(\sum_{i=1}^{\frac{n}{2}} \alpha_i(x_i).\overline{\delta_i}[x_i].(\mathtt{Eval}\ \gamma_i) + \sum_{i=\frac{n}{2}+1}^{n} \overline{\alpha_i}[x_i].\overline{\delta_i}.(\mathtt{Eval}\ \gamma_i)\right) \\
&)
\end{aligned}
$$

Figure 3: Encoding and evaluation of π-terms.

on the reserved channel γ_0. We have to prove that they are still equivalent under reduction. The proof use the standard properties of $\equiv$ given in Section 2.

Parallel Composition

$$
\begin{aligned}
& \mathbf{E}(P_1 \| P_2) \\
\equiv\ & (\nu\gamma_0)\Big((\nu abcde)(\overline{\gamma_0}[abcde].\overline{b}.(\nu\gamma_1\gamma_2)\overline{\gamma_0}[\gamma_1\gamma_2]. \\
& \qquad (\lceil P_1, \gamma_1 \rceil \| \lceil P_2, \gamma_2 \rceil)) \parallel (\mathtt{Eval}\ \gamma_0)\Big) \\
\dot{\approx}\ & (\nu\gamma_1\gamma_2)\Big((\lceil P_1, \gamma_1 \rceil \| \lceil P_2, \gamma_2 \rceil) \\
& \qquad \parallel (\mathtt{Eval}\ \gamma_1) \parallel (\mathtt{Eval}\ \gamma_2)\Big) \\
\equiv\ & \mathbf{E}(P_1) \| \mathbf{E}(P_2)
\end{aligned}
$$

Replication

$$
\begin{aligned}
& \mathbf{E}(!P) \\
\equiv\ & (\nu\gamma_0)\Big((\nu abcde)\overline{\gamma_0}[abcde].\overline{c}. \\
& \qquad !((\nu\gamma)\overline{\gamma_0}[\gamma].\lceil P, \gamma \rceil) \parallel (\mathtt{Eval}\ \gamma_0)\Big) \\
\dot{\approx}\ & \Big(!((\nu\gamma)\overline{\gamma_0}[\gamma].\lceil P, \gamma\rceil) \parallel !(\gamma_0(\gamma).(\mathtt{Eval}\gamma))\Big) \\
\dot{\approx}\ & \lceil P, \gamma_1 \rceil \parallel (\mathtt{Eval}\ \gamma_1) \parallel \lceil P, \gamma_2 \rceil \parallel (\mathtt{Eval}\ \gamma_2) \parallel \ldots \\
\equiv\ & \mathbf{E}(P) \| \mathbf{E}(P) \| \ldots \equiv !\mathbf{E}(P)
\end{aligned}
$$

Restriction

$$
\begin{aligned}
& \mathbf{E}((\nu x')P) \\
\equiv\ & (\nu\gamma_0)\Big((\nu abcde)\overline{\gamma_0}[abcde].\overline{d}. \\
& \qquad (\nu\gamma_1\gamma_2)\overline{\gamma_0}[\gamma_1\gamma_2].\gamma_1(x).\lceil P, \gamma_2 \rceil \parallel (\mathtt{Eval}\ \gamma_0)\Big) \\
\equiv\ & (\nu\gamma_0\gamma_1\gamma_2)\Big((\nu abcde)\overline{\gamma_0}[abcde].\overline{d}. \\
& \qquad \overline{\gamma_0}[\gamma_1\gamma_2].\gamma_1(x).\lceil P, \gamma_2 \rceil \parallel (\mathtt{Eval}\ \gamma_0)\Big) \\
\dot{\approx}\ & (\nu\gamma_1\gamma_2)\Big(\gamma_1(x).\lceil P, \gamma_2 \rceil \parallel (\nu x')\overline{\gamma_1}[x'].(\mathtt{Eval}\ \gamma_2)\Big)
\end{aligned}
$$

Note that in $\gamma_1(x)$, x is bound [9]. While there is no free occurrence of x in the first term of the parallel composition, (νx) may be factorized. It is easy to show that

$$
\begin{aligned}
& (\nu\gamma_1\gamma_2)\Big(\gamma_1(x).\lceil P, \gamma_2 \rceil \parallel (\nu x')\overline{\gamma_1}[x'].(\mathtt{Eval}\ \gamma_2)\Big) \\
\dot{\approx}\ & (\nu\gamma_1\gamma_2 x')\Big(\gamma_1(x).\lceil P, \gamma_2 \rceil \parallel \overline{\gamma_1}[x'].(\mathtt{Eval}\ \gamma_2)\Big) \\
\dot{\approx}\ & (\nu x)(\nu\gamma_2)\Big(\lceil P\{x \leftarrow x'\}, \gamma_2 \rceil \parallel (\mathtt{Eval}\ \gamma_2)\Big) \\
\dot{\approx}\ & (\nu x')\mathbf{E}(P\{x \leftarrow x'\})
\end{aligned}
$$

Sum

$$
\begin{aligned}
& \mathbf{E}(\textstyle\sum_{i=1}^{n} \pi_i.P_i) \\
\dot{\approx}\ & \left(\sum_{i=1}^{\frac{n}{2}} \delta_i(x_i).\lceil P_i, \gamma_i \rceil + \sum_{i=\frac{n}{2}+1}^{n} \delta_i.\lceil P_i, \gamma_i \rceil\right) \parallel \\
& \quad \Big(\quad \sum_{i=1}^{\frac{n}{2}} \alpha_i(x_i).\overline{\delta_i}[x_i].(\mathtt{Eval}\ \gamma_i) \\
& \qquad + \quad \sum_{i=\frac{n}{2}+1}^{n} \overline{\alpha_i}[x_i].\overline{\delta_i}.(\mathtt{Eval}\ \gamma_i) \quad \Big) \\
\equiv\ & S
\end{aligned}
$$

Thus, $\mathbf{E}(\sum_{i=1}^{n} \pi_i.P_i) \rightarrow^* S$ such that $S \downarrow_{\alpha_i}$, $1 \leq i \leq n$, and if S communicates on α_i then:

$$S \rightarrow \lceil P_i, \gamma_i \rceil \parallel (\mathtt{Eval}\ \gamma_i)$$

Additionally, for all α_i, $\sum_{i=1}^{n} \pi_i.\mathbf{E}(P_i) \downarrow_{\alpha_i}$ and may reduce to $\mathbf{E}(P_i)$. For all α, $\mathbf{E}(\sum_{i=1}^{n} \pi_i.P_i) \not\downarrow_\alpha$ since γ_0 is a local channel. This proves that

$$\mathbf{E}(\sum_{i=1}^{n} \pi_i.P_i) \dot{\approx} \sum_{i=1}^{n} \pi_i.\mathbf{E}(P_i)$$

□

Proposition 9 (Correctness of the interpreter)

$$\forall P \in \pi,\ P \dot{\approx} \mathbf{E}(P) \tag{8}$$

Proof
Immediate from Lemma 8 and by induction on the structure of P.

- if $P \equiv \mathbf{0}$ then $\forall \alpha,\ P \not\downarrow_\alpha$.

  $$\mathbf{E}(P) = (\nu\gamma_0)\Big((\nu abcde)(\overline{\gamma_0}[abcde].\overline{a} \parallel (\texttt{Eval}\ \gamma_0)\Big)$$

  Clearly, $\mathbf{E}(P) \not\downarrow_\alpha$ for all α since γ_0 is a local channel. Furthermore, $\mathbf{E}(P)$ reduces to $\mathbf{0}$ without being observable at α.

- if $P \equiv P_1 \| P_2$ then $P \downarrow_\alpha$ if and only if $P_1 \downarrow_\alpha$ or $P_2 \downarrow_\alpha$. By Lemma 8, $\mathbf{E}(P_1 \| P_2) \dot{\approx} \mathbf{E}(P_1) \| \mathbf{E}(P_2)$. So, $\mathbf{E}(P) \downarrow_\alpha$ iff $\mathbf{E}(P_1) \downarrow_\alpha$ or $\mathbf{E}(P_2) \downarrow_\alpha$. Induction completes the proof.

- if $P \equiv !P_1$ then $P \downarrow_\alpha$ iff $P_1 \downarrow_\alpha$ and by Lemma 8, $\mathbf{E}(!P) \dot{\approx} !\mathbf{E}(P)$.

- if $P \equiv (\nu x)P_1$ then $P \downarrow_\alpha$ iff $P_1 \downarrow_\alpha$ and by Lemma 8, $\mathbf{E}((\nu x)P_1) \dot{\approx} (\nu x)\mathbf{E}(P_1)$.

- if $P \equiv \sum_{i=1}^{n} \pi_i.P_i$ then $P \downarrow_{\alpha_i}$ and P may reduce to P_i. By Lemma 8, $\mathbf{E}(P) \dot{\approx} \sum_{i=1}^{n} \pi_i.\mathbf{E}(P_i)$. Thus, $\mathbf{E}(P) \downarrow_{\alpha_i}$. Furthermore by induction, if $\mathbf{E}(P)$ reduces to $\mathbf{E}(P_i)$, $\mathbf{E}(P_i) \downarrow_\alpha$ iff $P_i \downarrow_\alpha$.

□

5 Binding-time analysis

In this Section, we introduce a binding-time analysis (BTA) that determines which sub-expressions in a π-term are static, and which are dynamic. Note that in the calculus the only reduction rule concerns the sum operator. Thus, the analysis must consist of annotating which sums can be reduced at compile-time. Following [2], we introduce the notion of *two-level π-terms*.

Definition 10 (Two-level π-terms) *A two-level π-term is an expression generated by the following grammar:*

$$\begin{array}{rcl} P & ::= & \sum_{i=1}^{n} \pi_i.P_i \mid \underline{\sum}_{i=1}^{n} \pi_i.P_i \mid P \| Q \mid !P \mid (\nu x)P \\ \pi & ::= & \overline{\alpha}[x] \mid \alpha(x) \end{array}$$

An underlined sum indicates the dynamic behavior of the term. Such a dynamic term cannot be reduced at compile-time, because *none* of the prefixes π_i can communicate. Note that a sum $\sum_{i=1}^{n} \pi_i.P_i$ is static as soon as it reduces to P_i for some i, i.e. as soon as one of the communications π_i may be achieved statically.

The analysis of a process P requires some assumptions about which communications may be achieved between P and the processes Q_i running concurrently with P. These hypotheses are formalized by the notion of *BTA-context* Γ. Γ is an abstract description of the communications that are assumed to occur between P and the Q_i's during the analysis of P.

Definitions 11 (BTA-contexts and BTA-assertions) *Let P be a term to analyze. A BTA-context is a π-term Γ which indicates the communications that are assumed to occur between P and other processes during the analysis of P. Assertions of the form*

$$\Gamma \ \vdash P \ : \ \omega \tag{9}$$

state that, under the assumption Γ on the processes running concurrently with P, P may be annotated ω, where ω is a two-level π-term.

For instance, let us consider the term $P \equiv a(x).\mathbf{0} + b(y).\mathbf{0}$ and the BTA-assertions:

$$\mathbf{0} \ \vdash P \ : \quad a(x).\mathbf{0} \underline{+} b(y).\mathbf{0} \tag{10}$$

$$a(x) + b(y) \ \vdash P \ : \quad a(x).\mathbf{0} + b(y).\mathbf{0} \tag{11}$$

(10) states that the sum operator must be considered as dynamic without assumption. (11) states that assuming that $a(x) + b(y)$ is reduced by interference with another process, the sum operator in P may be static.

Intuitively, BTA-contexts are used to recursively analyze the sub-expressions of an initial term. Consider $P \equiv P_1 \| P_2$ and assume that $\Gamma_i \vdash P_i : \omega_i$, $1 \leq i \leq 2$. Γ_1 [resp. Γ_2] represents the hypotheses done when analyzing P_1 [resp. P_2]. Γ_i, $1 \leq i \leq 2$, is an abstraction of the processes which must be running concurrently with P_i in order to annotate it ω_i. If we consider now $P \equiv P_1 \| P_2$, then the only process running concurrently with P_1 [resp. P_2] is P_2 [resp. P_1]. Thus, the assumptions Γ_1 and Γ_2 are correct if they anneal each other, i.e. since Γ_1 and Γ_2 are π-terms, if $\Gamma \equiv \Gamma_1 \| \Gamma_2 \rightarrow^* \mathbf{0}$. If this last property is satisfied, ω may be partially evaluated since the static communications are these whose counterpart is present in P. A formal definition of correctness is provided by Definition 12.

Before introducing the inference system for the BTA, we define the following abbreviation. $\overline{\pi_i}$ represents the communication symmetric to π, i.e. $\overline{\alpha(x)} \equiv \overline{\alpha}[x]$ and $\overline{\overline{a}[x]} \equiv \alpha(x)$. Rules for the analysis are described in Figure 4.

The first rule states that one may annotate the π-term $\mathbf{0}$ by the two-level π-term $\mathbf{0}$, under the empty hypothesis $\mathbf{0}$. Rules concerning the parallel, duplicate and new operators just follow the structure of the term. The most important rules concern the sum operator. Consider that $\Gamma_i \vdash P_i : \omega_i$, $1 \leq i \leq n$. The first rule concerning $\sum$, states that the sum may be annotated static under the assumption that the term $\sum_{i=1}^{n} \pi_i.\Gamma_i$ can be reduced by interaction with the processes running concurrently with the process we analyze. Secondly, in order to underline (i.e. annotate dynamic) $\sum_{i=1}^{n} \pi_i.P_i$, we use the context

$$\Gamma \equiv (\sum_{i=1}^{n} \pi_i.\Gamma_i \| \sum_{i=1}^{n} \overline{\pi_i})$$

$$\overline{\mathbf{0} \vdash \mathbf{0} : \mathbf{0}}$$

$$\frac{\Gamma_1 \vdash P_1 : \omega_1 \quad \Gamma_2 \vdash P_2 : \omega_2}{\Gamma_1 \| \Gamma_2 \vdash P_1 \| P_2 : \omega_1 \| \omega_2}$$

$$\frac{\Gamma \vdash P : \omega}{!\Gamma \ \vdash !P : !\omega}$$

$$\frac{\Gamma \vdash P : \omega}{\Gamma \vdash (\nu x)P : (\nu x)\omega}$$

$$\frac{\Gamma_i \vdash P_i : \omega_i, \ 1 \le i \le n}{\sum_{i=1}^n \pi_i.\Gamma_i \vdash \sum_{i=1}^n \pi_i.P_i : \sum_{i=1}^n \pi_i.\omega_i}$$

$$\frac{\Gamma_i \vdash P_i : \omega_i, \ 1 \le i \le n}{\left(\sum_{i=1}^n \pi_i.\Gamma_i \| \sum_{i=1}^n \overline{\pi_i}\right) \vdash \sum_{i=1}^n \pi_i.P_i : \underline{\sum}_{i=1}^n \pi_i.\omega_i}$$

Figure 4: Binding-time analysis of π-terms.

Since each communication π_i may occur in Γ, this context states that for all i, $1 \le i \le n$, Γ_i may be a BTA-context for $\sum_{i=1}^n \pi_i.P_i$. Furthermore, Γ satisfies the property: $\Gamma \rightarrow^* \mathbf{0}$ iff $\forall i, \ 1 \le i \le n, \Gamma_i \rightarrow^* \mathbf{0}$. Thus, no assumption is done about the π_i's and the behavior of Γ only depends on the behavior of the Γ_i's.

For instance, let us consider the process

$$P \equiv a(x) + b(y) \ \| \ \overline{a}[u]$$

The analysis of P may yields the BTA-assertion

$$\Big(a(x) + b(y) \ \| \ \overline{a}[u]\Big) \ \vdash \ P \ : \ a(x) + b(y) \ \| \ \overline{a}[u] \qquad (12)$$

The BTA-assertion in equation (12) indicates that under the BTA-context $\overline{a}[x] + \overline{b}[y] \ \| \ a(u)$, both instructions $a(x) + b(y)$ and $\overline{a}[u]$ are static since a communication on the channel a may be executed at compile-time reducing P to $\mathbf{0}$.

Finally, to deal with the correctness of the partial evaluator with respect to annotated terms, we introduce the notion of well-annotatedness, used in the propositions of Section 6.

Definition 12 (Well-annotatedness) *A term P is well-annotated with annotation ω if $\Gamma \vdash P : \omega$, for some Γ such that $\Gamma \rightarrow^* \mathbf{0}$. In this case, we write $\vdash P : \omega$ instead of $\Gamma \vdash P : \omega$.*

6 Partial Evaluation

In this section, we show how to evaluate annotated terms. First, we give an encoding $\lfloor . \rfloor$ for the two-level π-terms. Second, we introduce the natural extension of `Eval` that yields a partial evaluator and prove its correctness with respect to the binding-time analysis and to the weak reduction equivalence relation. Similarly to `Eval`, `Pev` reads the input program on a reserved channel γ_0. In addition, `Pev` has to produce a $\lfloor . \rfloor$-encoded residual program. This is done using a second reserved channel. So, `Pev` reads the input program on a first channel μ_0, and outputs the encoding of the reduced term onto a second channel γ_0.

The encoding for annotated terms differs from $\lceil . \rceil$ only in the treatment of sum processes. As remarked in [3], the partial evaluator has to execute as many instructions in a given program as possible. Particularly, `Pev` may resolve non-determinism. Note that a sum process is annotated static as soon as at least one of the π_i's is static. From Equation (3), it is obvious that the choice is only made between static messages. This leads to the encoding function $\lfloor . \rfloor$ given in Figure 5. Reserved channels r, s, t, u, v, w are used instead of $a \ldots e$.

As indicated, two cases have to be distinguished when encoding communications.

Equations (13) and (14) only differ by the message $\overline{v}$ or $\overline{w}$ they send. $\overline{v}$ indicates that $\sum_{i=1}^n \pi_i.P_i$ is static, since at least one of the π_i's can communicate. $\overline{w}$ is sent when the communications cannot be computed at compile-time, and must be encoded as residual code.

The partial evaluator `Pev`, given in Figure 6, has to deal with two kinds of instructions. When a communication π is annotated static, π is executed. Otherwise, `Pev` produces the encoding of the residual program $\lceil \pi \rceil$, and recursively partially evaluates the continuation of the program.

We present now the most natural extension of `Eval` to a partial evaluator, denoted `Pev`, and prove its correctness. Let $\lfloor \omega \rfloor$ be an annotated input π-term to specialize. Similarly to Section 4, recursion is left implicit. `Pev` has to execute communications annotated static in P, and to produce the encoding $\lceil R \rceil$ of the residual program R.

First of all, since $\lfloor \omega \rfloor$ has to be encoded on a particular channel, `Pev` takes two arguments. The first one, μ_0, indicates the location of the encoding of the annotated input program P. The second, γ_0, locates the channel used to encode the output residual program R. Equation (15), in Figure 6, describes the structure of `Pev`.

In the π-term `A`, the residual code generated on γ_0, when the message r is encountered, is the encoding of the null process.

In order to partially evaluate the parallel composition of P_1 and P_2, `Pev` creates two new channels γ_1 and γ_2 as well as channels a to e (π-term `B`). Locations μ_1 and μ_2 of the continuations are received on μ_0, and P_1 and P_2 are partially evaluated.

Concerning the replication operator (π-term `C`), an occurrence of message t yields `Pev` to produce a variant of the encoding function $\lceil . \rceil$, in which continuations are partially evaluated.

When the encoding of (νx) is encountered on μ_0, the encoding of the restriction operator is transmitted to the evaluator. `Eval` sends the new name to `Pev` which transmits it to the encoding process in order to achieve the specialization.

Concerning the sum, we had two choices (13) and (14) previously. In (13), v indicates the occurrence of a static communication. The action is executed and the continuation is partially evaluated. `E` is similar to the corresponding case of `Eval`, except that (`Pev` μ_i γ_0) is substituted to (`Eval` γ_i). When, in (14), a message w is encountered, the dynamic communications are encoded on γ_0 and `Pev` is applied to the continuations.

In order to deal with the correctness of `Pev`, we introduce the following notations where ω represents a two-level π-term:

$$\mathbb{P}(\omega, \gamma_0) \equiv (\nu \mu_0)\Big(\lfloor \omega, \mu_0 \rfloor \ \| \ (\texttt{Pev} \ \mu_0 \ \gamma_0)\Big) \qquad (16)$$

$$\lfloor \mathbf{0}, \mu_0 \rfloor = (\nu rstuvw)\overline{\mu_0}[rstuvw].\overline{r}$$

$$\lfloor P_1 \| P_2, \mu_0 \rfloor = (\nu rstuvw)\overline{\mu_0}[rstuvw].\overline{s}.(\nu \mu_1 \mu_2)\overline{\mu_0}[\mu_1 \mu_2].(\lfloor P_1, \mu_1 \rfloor \| \lfloor P_2, \mu_2 \rfloor)$$

$$\lfloor !P, \mu_0 \rfloor = (\nu rstuvw)\overline{\mu_0}[rstuvw].\overline{t}.!((\nu \mu)\overline{\mu_0}[\mu].\lfloor P, \mu \rfloor)$$

$$\lfloor (\nu x)P, \mu_0 \rfloor = (\nu rstuvw)\overline{\mu_0}[rstuvw].\overline{u}.(\nu \mu_1 \mu_2)\overline{\mu_0}[\mu_1 \mu_2].\mu_1(x).\lfloor P, \mu_2 \rfloor$$

$$\begin{aligned}\lfloor \textstyle\sum_{i=1}^{n} \pi_i.P_i, \mu_0 \rfloor = \; & (\nu rstuvw)\overline{\mu_0}[rstuvw].\overline{v}. \\ & (\nu \mu_1 \ldots \mu_n \eta_1 \ldots \eta_n)\overline{\mu_0}[\mu_1 \ldots \mu_n \eta_1 \ldots \eta_n \alpha_1 \ldots \alpha_n x_1 \ldots x_n]. \\ & \Big(\textstyle\sum_{i=1}^{\frac{n}{2}} \eta_i(x_i).\lfloor P_i, \mu_i \rfloor + \sum_{i=\frac{n}{2}+1}^{n} \eta_i.\lfloor P_i, \mu_i \rfloor \Big)\end{aligned} \tag{13}$$

$$\begin{aligned}\lfloor \underline{\textstyle\sum}_{i=1}^{n} \pi_i.P_i, \mu_0 \rfloor = \; & (\nu rstuvw)\overline{\mu_0}[rstuvw].\overline{w}. \\ & (\nu \mu_1 \ldots \mu_n \eta_1 \ldots \eta_n)\overline{\mu_0}[\mu_1 \ldots \mu_n \eta_1 \ldots \eta_n \alpha_1 \ldots \alpha_n x_1 \ldots x_n]. \\ & \Big(\textstyle\sum_{i=1}^{\frac{n}{2}} \eta_i(x_i).\lfloor P_i, \mu_i \rfloor + \sum_{i=\frac{n}{2}+1}^{n} \eta_i.\lfloor P_i, \mu_i \rfloor \Big)\end{aligned} \tag{14}$$

Figure 5: Encoding of two-level π-terms.

$$\mathtt{Pev}(\mu_0, \gamma_0) \equiv \mu_0(rstuvw).(\mathtt{A} + \mathtt{B} + \mathtt{C} + \mathtt{D} + \mathtt{E} + \mathtt{F}) \tag{15}$$

$$\mathtt{A} \equiv r.(\nu abcde)\overline{\gamma_0}[abcde].\overline{a} \equiv r.\lceil \mathbf{0}, \gamma_0 \rceil$$

$$\begin{aligned}\mathtt{B} \equiv \; & s.\mu_0(\mu_1\mu_2).\overbrace{(\nu abcde)\overline{\gamma_0}[abcde].\overline{b}.(\nu \gamma_1 \gamma_2)\overline{\gamma_0}[\gamma_1\gamma_2].}^{\text{Beginning of the encoding } \lceil P_1 \| P_2, \gamma_0 \rceil} \\ & \underbrace{\Big((\mathtt{Pev}\ \mu_1\ \gamma_1) \parallel (\mathtt{Pev}\ \mu_2\ \gamma_2) \Big)}_{(\mathtt{Pev}\ \mu_i\ \gamma_i) \text{ substituted to } \lceil P_i, \gamma_i \rceil \text{ in } \lceil P_1, \gamma_1 \rceil \parallel \lceil P_2, \gamma_2 \rceil,\ i=1,2}\end{aligned}$$

$$\mathtt{C} \equiv t.\overbrace{(\nu abcde).\overline{\gamma_0}[abcde].\overline{c}}^{\text{Beginning of } \lceil !P, \gamma_0 \rceil}.\underbrace{!\Big(\mu_0(\mu).(\nu\gamma)\overline{\gamma_0}[\gamma].(\mathtt{Pev}\ \mu\ \gamma) \Big)}_{\text{Addition of } \mu_0(\mu) \text{ and } (\mathtt{Pev}\ \mu\ \gamma) \text{ substituted to } \lceil P, \gamma \rceil}$$

$$\mathtt{D} \equiv u.\mu_0(\mu_1\mu_2).\underbrace{(\nu abcde)\overline{\gamma}_0[abcde].\overline{d}.(\nu \gamma_1 \gamma_2)\overline{\gamma}_0[\gamma_1 \gamma_2].\gamma_1(x).\overline{\mu_1}[x].(\mathtt{Pev}\ \mu_2\ \gamma_2)}_{\overline{\mu_1}[x].(\mathtt{Pev}\ \mu_2\ \gamma_2) \text{ substituted to } \lceil P, \gamma_2 \rceil \text{ in } \lceil (\nu x)P, \gamma_0 \rceil}$$

$$\begin{aligned}\mathtt{E} \equiv \; & v.\mu_0(\mu_1 \ldots \mu_n \eta_1 \ldots \eta_n \alpha_1 \ldots \alpha_n x_1 \ldots x_n). \\ & \Big(\textstyle\sum_{i=1}^{\frac{n}{2}} \alpha_i(x_i).\overline{\eta_i}[x_i].(\mathtt{Pev}\ \mu_i\ \gamma_0) + \sum_{i=\frac{n}{2}+1}^{n} \overline{\alpha_i}[x_i].\overline{\eta_i}.(\mathtt{Pev}\ \mu_i\ \gamma_0) \Big)\end{aligned}$$

$$\begin{aligned}\mathtt{F} \equiv \; & w.\mu_0(\mu_1 \ldots \mu_n \eta_1 \ldots \eta_n \alpha_1 \ldots \alpha_n x_1 \ldots x_n). \\ & \underbrace{(\nu abcde)\overline{\gamma_0}[abcde].\overline{e}.(\nu \gamma_1 \ldots \gamma_n \delta_1 \ldots \delta_n)\overline{\gamma_0}[\gamma_1 \ldots \gamma_n \delta_1 \ldots \delta_n \pi_1 \ldots \pi_n].}_{\text{Beginning of the encoding } \lceil \sum_{i=1}^{n} \pi_i.P_i, \gamma_0 \rceil} \\ & \Big(\textstyle\sum_{i=1}^{\frac{n}{2}} \delta_i(x_i).\overline{\eta_i}[x_i].(\mathtt{Pev}\ \mu_i\ \gamma_i) + \sum_{i=\frac{n}{2}+1}^{n} \delta_i.\overline{\eta_i}.(\mathtt{Pev}\ \mu_i\ \gamma_i) \Big)\end{aligned}$$

Figure 6: Partial evaluation of two-level π-terms.

$$\mathbf{P}(\omega) \equiv (\nu\gamma_0)\Big(\mathbb{P}(\omega,\gamma_0) \parallel (\texttt{Eval}\ \gamma_0)\Big) \tag{17}$$

The term $\mathbb{P}(\omega,\gamma_0)$ is the parallel composition of the encoding of the annotated term ω on μ_0 and of $(\texttt{Pev}\ \mu_0\ \gamma_0)$. It represents the partial evaluation of a term P, annotated ω. The residual code is sent onto γ_0. $\mathbf{P}(\omega)$ is the parallel composition of $\mathbb{P}(\omega,\gamma_0)$ and $(\texttt{Eval}\ \gamma_0)$, i.e. the concurrent execution of three terms, the encoding, the partial evaluator, and the interpreter. We prove the correctness of the partial evaluation by comparing the behaviors of $\mathbf{P}(\omega)$ and P. Note that we only consider well-annotated terms. No warranty is given for partial evaluation correctness of non well-annotated terms.

Lemma 13 (Compositionality of P) *Let P be a π-term and ω a well-annotated two-level π-term associated to P. Then, the following properties are verified:*

(i) $\omega \equiv \omega_1 \| \omega_2 \Rightarrow \mathbf{P}(\omega) \dot{\approx} \mathbf{P}(\omega_1) \| \mathbf{P}(\omega_2)$.

(ii) $\omega \equiv !\omega_1 \Rightarrow \mathbf{P}(\omega) \dot{\approx}\ !\mathbf{P}(\omega_1)$.

(iii) $\omega \equiv (\nu x)\omega_1 \Rightarrow \mathbf{P}(\omega) \dot{\approx} (\nu x)\mathbf{P}(\omega_1)$.

(iv) $\omega \equiv \sum_{i=1}^n \pi_i.P_i \Rightarrow \mathbf{P}(\omega) \dot{\approx} \sum_{i=1}^n \pi_i.\mathbf{P}(\omega_i)$, *the sum being either static or dynamic.*

PROOF

The proof is by induction on the structure of P. Cases $\omega \equiv \omega_1 \| \omega_2$, $\omega \equiv !\omega_1$ and $\omega \equiv (\nu x)\omega_1$ are similar to those of Lemma 8. For sum processes two cases have to be distinguished. The difference appears in Equations (19) and (21). Note in both cases the location of the communication π_i (recall that π_i denotes $\alpha_i(x_i)$ or $\overline{\alpha}_i[x_i]$).

- if $\omega = \underline{\sum}_{i=1}^n \pi_i.P_i$, i.e. all communications are dynamic, we have

$$\begin{array}{ll} & \lfloor \omega, \mu_0 \rfloor \\ \equiv & (\nu rstuvw)\overline{\mu_0}[rstuvw].\overline{w}.(\nu\mu_1\ldots\mu_n\eta_1\ldots\eta_n) \\ & \overline{\mu_0}[\mu_1\ldots\mu_n\eta_1\ldots\eta_n\alpha_1\ldots\alpha_n x_1\ldots x_n]. \\ & \Big(\sum_{i=1}^{\frac{n}{2}} \eta_i(x_i).\lfloor P_i,\mu_i\rfloor + \sum_{i=\frac{n}{2}+1}^n \eta_i.\lfloor P_i,\mu_i\rfloor\Big)\end{array}$$

as the annotated term. Consequently

$$\begin{array}{ll} & \mathbf{P}(\omega) \\ \equiv & (\nu\gamma_0 rstuvw\mu_0) \\ & \Big(\lfloor \omega,\mu_0\rfloor \parallel (\texttt{Pev}\ \mu_0\ \gamma_0) \parallel (\texttt{Eval}\ \gamma_0)\Big)\end{array} \tag{18}$$

$$\begin{array}{ll} \dot{\approx} & \sum_{i=1}^{\frac{n}{2}} \eta_i(x_i).\lfloor P_i,\mu_i\rfloor \\ & \parallel \sum_{i=\frac{n}{2}+1}^n \eta_i.\lfloor P_i,\mu_i\rfloor \\ & \parallel \sum_{i=1}^{\frac{n}{2}} \delta_i(x_i).\overline{\eta_i}[x_i].(\texttt{Pev}\ \mu_i\ \gamma_i) \\ & \parallel \sum_{i=\frac{n}{2}+1}^n \delta_i.\overline{\eta_i}.(\texttt{Pev}\ \mu_i\ \gamma_i) \\ & \parallel \sum_{i=1}^{\frac{n}{2}} \alpha_i(x_i).\overline{\delta_i}[x_i].(\texttt{Eval}\ \gamma_i) \\ & \parallel \underbrace{\sum_{i=\frac{n}{2}+1}^n \overline{\alpha_i}[x_i].\overline{\delta_i}.(\texttt{Eval}\ \gamma_i)}_{\pi_i \text{ is a prefix of } \texttt{Eval}}\end{array} \tag{19}$$

Call P' the term defined by (19). We have $P' \downarrow_{\alpha_i}$, $1 \le i \le n$. Assume now that P' communicates on α_i.

$$\begin{array}{lll} P' & \dot{\approx} & \sum_{i=1}^n \lfloor P_i,\mu_i\rfloor \parallel \sum_{i=1}^n (\texttt{Pev}\ \mu_i\ \gamma_i) \parallel (\texttt{Eval}\ \gamma_i) \\ & \dot{\approx} & \mathbf{P}(\omega_i)\end{array}$$

On the other hand, $\sum_{i=1}^n \pi_i.\mathbf{P}(\omega_i) \downarrow_{\alpha_i}$ and also reduces to $\mathbf{P}(\omega_i)$ after communication on α_i. Furthermore, the term ω_i occuring in $\mathbf{P}(\omega_i)$ is well-annotated since the annotation $\big(\sum_{i=1}^n \pi_i.\Gamma_i \| \sum_{i=1}^n \overline{\pi_i}\big)$ ensures that

$$(\Gamma \rightarrow^* \mathbf{0}) \iff (\forall i,\ 1 \le i \le n,\ \Gamma_i \rightarrow^* \mathbf{0})$$

- if $\omega = \sum_{i=1}^n \pi_i.P_i$, i.e. some communications are static, we have

$$\begin{array}{ll} & \lfloor \omega, \mu_0 \rfloor \\ \equiv & (\nu rstuvw)\overline{\mu_0}[rstuvw].\overline{v}. \\ & \quad (\nu\mu_1\ldots\mu_n)\overline{\mu_0}[\mu_1\ldots\mu_n\alpha_1\ldots\alpha_n x_1\ldots x_n]. \\ & \quad \sum_{i=1}^n \lfloor P_i,\mu_i\rfloor\end{array}$$

as the annotated term. Again

$$\begin{array}{ll} & \mathbf{P}(\omega) \\ \equiv & (\nu\gamma_0 rstuvw\mu_0) \\ & \Big(\lfloor \omega,\mu_0\rfloor \parallel (\texttt{Pev}\ \mu_0\gamma_0) \parallel (\texttt{Eval}\ \gamma_0)\Big)\end{array} \tag{20}$$

$$\begin{array}{ll} \dot{\approx} & \sum_{i=1}^{\frac{n}{2}} \eta_i(x_i).\lfloor P_i,\mu_i\rfloor \\ & \parallel \sum_{i=\frac{n}{2}+1}^n \eta_i.\lfloor P_i,\mu_i\rfloor \\ & \parallel \sum_{i=1}^{\frac{n}{2}} \alpha_i(x_i).\overline{\eta_i}[x_i].(\texttt{Pev}\ \mu_i\ \gamma_i) \\ & \parallel \underbrace{\sum_{i=\frac{n}{2}+1}^n \overline{a_i}[x_i].\overline{\eta_i}.(\texttt{Pev}\ \mu_i\ \gamma_i)}_{\pi_i \text{ is a prefix of } \texttt{Pev}} \\ & \parallel (\texttt{Eval}\ \gamma_0)\end{array} \tag{21}$$

Write P' for the term in (21). $\sum_{i=1}^n \pi_i.\mathbf{P}(\omega_i) \downarrow_{\alpha_i}$, $1 \le i \le n$ and $\mathbf{P}(\omega) \dot{\approx} P'$ such that $P' \downarrow_{\alpha_i}$, $1 \le i \le n$. We have to show that this property is still verified under reduction.

Since $\sum_{i=1}^n \pi_i.P_i$ is annotated static, the process $\Psi \equiv \sum_{i=1}^n \pi_i.\Gamma_i$ occurs in Γ. Since ω is well-annotated, $\Gamma \vdash P : \omega$ for some Γ such that $\Gamma \rightarrow^* \mathbf{0}$. Thus

$$\Gamma \equiv \Psi \parallel \Psi'$$

and such that $\Psi' \vdash P' : \omega'$ and $P' \downarrow_{\overline{\alpha}_i}$ for some i, $1 \le i \le n$. Therefore, P communicates on α_i. The partial evaluation process is not blocked and we obtain

$$\begin{array}{lll} \mathbf{P}(\omega) & \dot{\approx} & \sum_{i=1}^n \lfloor P_i,\mu_i\rfloor \parallel (\texttt{Pev}\ \mu_i\ \gamma_0) \parallel (\texttt{Eval}\ \gamma_0) \\ & \dot{\approx} & \mathbf{P}(\omega_i)\end{array}$$

which is structurally equivalent to $\sum_{i=1}^n \pi_i.\mathbf{P}(\omega_i)$ after communication on π_i. Additionally ω_i is well-annotated since $\Gamma \rightarrow \Gamma_i$ if we consider that a communication on α_i occurs.

□

Proposition 14 (Correctness of partial evaluation) *Let P be a π-term. The following property holds:*

$$\vdash P : \omega \Rightarrow \mathbf{P}(\omega) \approx P \quad (22)$$

Remember that $\vdash P : \omega$ means that $\Gamma \vdash P : \omega$ for some Γ such that $\Gamma \rightarrow^* \mathbf{0}$. Proposition 14 states that if a π-term P is annotated ω under assumption Γ, and if $\Gamma \rightarrow^* \mathbf{0}$, then the evaluation using `Eval` of the residual program produced by partial evaluation of ω is weakly equivalent to the original program P.

PROOF

The proof is by induction on the structure of the annotated term ω and is similar to the one of Proposition 9. The hypothesis $\vdash P : \omega$ allows the use of Lemma 13.

□

7 Discussion

First, note we have considered the *asynchronous* π-calculus. Otherwise, if we consider synchronous communications, `Pev` may be a *lazy* partial evaluator in the sense that continuations are recursively partially evaluated only when required. For instance, let us examine the parallel composition of two processes P_1 and P_2. If communications on γ_0 are synchronous, the process is stopped as long as `Eval`(γ_0) is not present. Partial evaluations of P_1 and P_2 will be done only when required by `Eval`(γ_0). The same behavior arises for $!P$ and $(\nu x)P$. When a dynamic communication π is encountered, its encoding is sent onto γ_0 and the partial evaluation of the continuation will be deferred until π is evaluated. In this last case, *only* the continuation P_i of the selected term is partially evaluated. P_j, $j \neq i$ is never specialized.

However, `Pev` sketches the scheme of partial evaluation for a parallel language and any implementation of a specializer based on it would enforce continuations to be computed statically. In addition, `Pev` satisfies the property

$$P \rightarrow^* \mathbf{0} \Rightarrow \mathbf{P}(P) \rightarrow^* \mathbf{0} \quad (23)$$

This means that `Pev` does not generate useless processes that may stay in the environment after partial evaluation. It is possible to write an *eager* partial evaluator for π-calculus by computing the continuations *concurrently* to the communications on γ_0 and μ_0. For instance `B` would roughly be of the form

$$\begin{aligned} \mathtt{B} \;\equiv\;& s.\mu_0(\mu_1\mu_2).(\nu abcde)\overline{\gamma_0}[abcde].\overline{b}.(\nu\gamma_1\gamma_2) \\ & \left(\overline{\gamma_0}[\gamma_1\gamma_2] \;\|\; (\mathtt{Pev}\ \mu_1\ \gamma_1) \;\|\; (\mathtt{Pev}\ \mu_2\ \gamma_2)\right) \end{aligned}$$

with respect to the new encoding:

$$\begin{aligned} \lfloor P_1 \| P_2, \mu_0 \rfloor \;=\;& (\nu rstuvw)\overline{\mu_0}[rstuvw].\overline{s}.(\nu\mu_1\mu_2) \\ & (\overline{\mu_0}[\mu_1\mu_2] \;\|\; \lfloor P_1, \mu_1 \rfloor \;\|\; \lfloor P_2, \mu_2 \rfloor) \end{aligned}$$

However this eager partial evaluator may generate processes that will never be reduced. These processes are unobservable since they communicates on reserved channels that nobody else knows of. Therefore, proofs of correctness are still valid since they are based on observability. Equation (23) becomes:

$$P \rightarrow^* \mathbf{0} \Rightarrow \mathbf{P}(P) \rightarrow^* Q \text{ such that } Q \dot{\approx} \mathbf{0}$$

8 Conclusion

In this paper, we discussed self-applicable partial evaluation for *pure* π-calculus. This language models concurrent behavior of parallel and object-oriented systems. We used the three-steps methodology consisting of writing a meta-interpreter, introducing an abstract analysis and exhibiting a self-applicable partial evaluator. We proved the correctness of `Pev` w.r.t. the weak reduction equivalence.

First of all, let us remark that every operator in the language is required in order to obtain reflection. A self-interpreter cannot be written for a strict subset of the π-calculus. Parallel composition allows application of the evaluator to the encoding of the terms. Replication is needed for recursion. Restriction is required for observability, and sums allow conditional choices.

Second, note that `Eval` is a parallel interpreter. In addition, our approach is compatible with Futamura's projections. Thus, a compiler `Comp` can be automatically generated. `Comp` compiles parallel programs, and the compilation process itself is parallel. In addition, `Comp`'s correctness is a corollary of `Eval`'s and `Pev`'s ones. A compiler generator `Cogen` can also be produced. Its inputs is a parallel interpreter, i.e. the parallel semantics of any language. It is transformed into a parallel compiler.

Now, our considerations focus on an implementation yielding `Cogen`. This implementation could be achieved using Pict [15], a language based on π-calculus. However, this is not clear that Pict's strong type system will be well suited for reflection.

Finally, we believe that the principles of `Pev` could successfully be applied to the kernel of a real concurrent language with distinct variables and constants. Difficulties are to define such a reflective language. In addition, the BTA should produce two different kinds of informations. First, the static communications that may be executed at compile-time. In this context, variables are unknown either if they depend on a dynamic expression or if their assignments are related to dynamic communications.

References

[1] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[2] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[3] Haruo Hosoya, Naoki Kobayashi, and Akinori Yonezawa. Partial evaluation scheme for concurrent languages and its correctness. In *Euro-par'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 625–632. Springer-Verlag, 1996.

[4] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communications for asynchronous concurrent programming languages. In *SAS'95*, volume 983 of *Lecture Notes in Computer Science*, pages 225–242. Springer-Verlag, 1995.

[5] Peter Lee and Mark Leone. Deferred compilation : The automation of run-time code generation. Technical

report, School of Computer Science, Carnegie Mellon University, Pittsburgh, December 1993.

[6] Murakami Masaki. Partial evaluation of reactive communicating processes using temporal logic formulas and its applications. In *Proc. of Workshop on Algebraic and Object-Oriented Approaches to Software Science*, 1995.

[7] Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. Implementing parallel language constructs using a reflective object-oriented language. In *Reflexion'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 625–632. Springer-Verlag, 1996.

[8] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[9] Robin Milner. The polyadic π-calculus: a tutorial. October 1991.

[10] Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, July 1992.

[11] Torben Æ. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 116–121. New Haven, CT: Yale University, 1992.

[12] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–363, July 1993.

[13] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Programming Language Design and Implementation, Atlanta, Georgia*, 1988.

[14] Frank Pfenning and Peter Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, 89:137–159, 1991.

[15] Benjamin C. Pierce and David N. Turner. *Pict Language Definition, Version 3.9d*, 1996.

[16] David Walker. Objects and the pi-calculus. *Information and Computation*, 1995.

[17] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993.