Partial Evaluation of Concurrent Programs

Matthieu Martel[1] and Marc Gengler[2]

[1] CEA - Recherche Technologique
LIST-DTSI-SLA, 91191 Gif-sur-Yvette cedex, France
mmartel@cea.fr
[2] ESIL – Laboratoire d'Informatique de Marseille
163, avenue de Luminy, Case 925, 13288 Marseille cedex 9, France
Marc.Gengler@esil.univ-mrs.fr

Abstract. In this article, we introduce a partial evaluator for a concurrent functional language with synchronous communications over channels, dynamic process and channel creations, and the ability to communicate channel names. Partial evaluation executes at compile-time the communications of a program for which the emitter, the receptor and the message contents are statically known. The partial evaluator and the static analyses used to guide it were implemented and we show the results of the specialization of concurrent programs for particular execution contexts, corresponding to different assumptions on the network or on the messages.

Keywords: Partial Evaluation, Concurrent Languages, Binding-time Analysis, Control Flow Analysis.

1 Introduction

Partial evaluation is a technique used to execute a program for which only one part of the data is known [10]. The *static* instructions, depending on the known data, are executed, while the *dynamic* instructions, depending at least partly on unknown data, are frozen. We obtain a *residual* or *specialized* program, made of the pieces of code which could not be executed and of the results of those parts of the computation that could be executed. In order to determine the parts of a program which can be executed because they solely depend on known data, partial evaluators (PE) usually use the results of a static analysis called *binding-time analysis* (BTA) [3,10].

In this article, we are interested in partial evaluation of a concurrent language with synchronous communications over channels, dynamic process creation and the ability to communicate channel names and functions. Sequential parts of the programs are written in a functional style. For this kind of programs, partial evaluation allows the static execution of those communications for which the emitter, the receptor and the contents of the message are known [6,9,11]. We obtain a residual program with fewer communications than the original one. For instance, partial evaluation has been used to scale up a commercial version of the RPC protocol, yielding up to 3.75 times faster code [14]. However, since no specific techniques were used to specialize the communication primitives, the partial evaluation was done by means of run-time specialization techniques which could have been avoided by using an adequate method for partial evaluation of the communications.

We introduce a partial evaluator, denoted `Pev` which uses the result of a BTA and of a control flow analysis (CFA) [16,19] in two different ways. The analyses are used, first, to determine the functions possibly called at a given application point and, second, to determine the possible synchronizations of the program, i.e. the possible matching pairs of emitters and receptors. The CFA of [13], the BTA and the partial evaluator were implemented and we describe some experiments. We show how to specialize, by partial evaluation, concurrent programs with respect to particular execution contexts, considering for instance static knowledge on the topology of the network, or assumptions on the behavior of the network, or assumptions on the data transmitted. In all cases, we show that `Pev` reduces away all communications that rely only on the knowledge available and outputs residual programs with less communications than the original ones.

Even though partial evaluation techniques have been widely studied for sequential languages, partial evaluation of concurrent programs has received little attention. However, concurrency introduces new concepts in programming languages for which specific methods must be developed. Hosoya *et al.* have proposed an *on-line* PE for a concurrent language close to the one treated in this article [9]. On-line PE do not use the results of a BTA in order to improve their accuracy. Marinescu and Goldberg have proposed a PE for a CSP-like language with static channels [11]. The authors have proposed a PE for the π-calculus as well as sufficient conditions on the annotations to ensure the correctness of the residual program wrt. the original one [6,12]. Solberg *et al.* and Bodei *et al.* have proposed control flow analyses (CFA) which can be used to improve the precision of the BTA [2,20]. Improvements are proposed in [12,13].

This article is organized as follows. Section 2 gives the principles of the binding time analysis used to annotate the programs provided to the partial evaluator. Section 3 introduces the partial evaluator `Pev` and Section 4 presents and discusses the specialization obtained with `Pev` on some examples of concurrent programs.

2 Program Analysis

A partial evaluator uses the annotations attached to an input program p to determine how to specialize it. These annotations are the result of static analyses of p. Among these, a binding-time analysis of p determines which instructions depend on the static data and can be executed by the PE. For concurrent languages with explicit communications, the BTA has to determine which communications are static, i.e. occur on channels known at partial evaluation-time and transmit static data. Static communications are executed at partial evaluation-time while dynamic ones are left unchanged in the residual program output by the PE.

In order to produce a precise annotation of the program, the BTA itself uses the annotations given by an analysis of the topology of the communications which indicates the pairs of possibly matching emitters and receptors. In our implementation, the topology is computed by the control flow analysis described in [13]. This CFA builds a reduced product automaton, which is polynomial in size and describes an approximation of all possible interleavings of the program. This automaton is used to approximate the possible communications, independently of their relative ordering. Indeed, the exact position of a matching pair within the execution trace is not relevant in our context.

The BTA we use is formally defined in [12]. Concerning the sequential part of the language, it is a usual BTA based on the one introduced by Bondorf and Jorgensen [3], for instance. Concerning concurrency primitives, it uses the topological information provided by the CFA previously presented and discussed. In the following, we illustrate its specific features using some examples.

As indicated above, the precision of the BTA depends on the one of the topology provided by the CFA. Let us consider a program with two processes which realize first a static and next a dynamic communication on the same channel γ. The results produced by our BTA for this program are given in Equation (1), in which underlined operations are dynamic.

$$\texttt{let}\ p_1 = \texttt{fork}\begin{pmatrix}\texttt{let}\ s_1 = \texttt{send}\ \gamma\ S\ \texttt{in}\\ \texttt{let}\ s_2 = \underline{\texttt{send}}\ \gamma\ D\ \texttt{in}\\ \ldots\end{pmatrix}\ \texttt{in}\ \begin{pmatrix}\texttt{let}\ r_1 = \texttt{receive}\ \gamma\ \texttt{in}\\ \texttt{let}\ r_2 = \underline{\texttt{receive}}\ \gamma\ \texttt{in}\\ \ldots\end{pmatrix} \tag{1}$$

We observe that only the second communication is annotated as being dynamic. This is due to the fact that the topological information allows, in this case, to determine the exact pairs of emitters and receptors [13]. A less precise BTA, for instance based on the topological information provided by the analysis used in [20], would annotate both communications as being dynamic since γ is at least once used to communicate a dynamic value.

A second aspect concerns the emission and reception primitives for which the channel and the contents of the message (for receptions) are static, but which cannot synchronize because there is no matching communication in the program. For instance, let us consider an emission **send** e_0 e_1 with e_0 and e_1 static. Such a communication point is annotated static by the analysis, leading the partial evaluator into a blocking state. This is however correct, since the original program would block in exactly the same way during a usual execution. This problem is in fact comparable to the problem of static infinite loops in partial evaluation of sequential programs.

Next, certain communications with static parameters must nevertheless be annotated as being dynamic, due to the context. This happens for instance in the program of Equation (2), in which a reception may synchronize with two different emissions, depending on a dynamic condition. In this case, the execution of the communication must be delayed, because the control is not known.

$$\texttt{let}\ p_1 = \texttt{fork}\begin{pmatrix}\underline{\texttt{if}}\ \underline{\mathit{cond}}\ \underline{\texttt{then}}\ \underline{\texttt{send}}\ \gamma\ 0\\ \underline{\texttt{else}}\ \underline{\texttt{send}}\ \gamma\ 1\end{pmatrix}\ \texttt{in}\ \underline{\texttt{receive}}\ \gamma \tag{2}$$

The creation of a new process is handled in a way similar to communications. A `fork` is static as long as it does not occur within a dynamic context (for example the conditional of Equation 2, or a dynamic loop). In this case, the process creation is frozen and all communications occurring in the code of the new process become dynamic.

3 Partial Evaluation

The language used by our PE is an untyped subset of Concurrent ML [1,18] based on the language λ_{cv} defined by Reppy in [17]. However, the technique described here does not depend on the functional nature of the language but only on the underlying model of communication that Concurrent ML supports. The choice of an untyped functional language is motivated by the fact that it makes self-application of the PE possible and allows to automatically generate, using the Futamura's projections [5], a compiler generator, as shown in [12]. λ_{cv} is a language with dynamic process and channel creations and synchronous communications over channels. Channel names created by the instruction `channel()` and functions are ground values which may be communicated. The basic syntax of Concurrent ML is defined by the first two lines of the grammar given in Equation (3). The third and fourth lines are introduced later.

$$\begin{array}{rl} e ::= & c \mid x \mid \texttt{fun}\ x \texttt{ => } e_0 \mid \texttt{rec}\ f\ x \texttt{ => } e_0 \mid e_0\ @\ e_1 \mid \texttt{if}\ e_0\ e_1\ e_2 \\ & \mid \texttt{channel()} \mid \texttt{fork}\ e_0 \mid \texttt{send}\ e_0\ e_1 \mid \texttt{receive}\ e_0 \\ & \mid \underline{c} \mid \underline{\texttt{fun}}\ x \texttt{ => } e_0 \mid \underline{\texttt{rec}}\ f\ x \texttt{ => } e_0 \mid e_0\ \underline{@}\ e_1 \mid \underline{\texttt{if}}\ e_0\ e_1\ e_2 \\ & \mid \underline{\texttt{channel()}} \mid \underline{\texttt{fork}}\ e_0 \mid \underline{\texttt{send}}\ e_0\ e_1 \mid \underline{\texttt{receive}}\ e_0 \mid \texttt{lift}\ e \end{array} \tag{3}$$

The language contains conditionals and the operator `rec` for recursive functions. `channel()` denotes a function call which creates and returns a new channel name k, different from all the existing ones. `fork` e_0 creates a new process which computes e_0. `send` e_0 e_1 is the emission of the value of e_1 on the channel resulting from the evaluation of e_0. e_0 and e_1 respectively are the *subject* and the *object* of the communication. `receive` e_0 is the reception of a value on the channel name described by e_0 (the subject of the reception). Values are in the domains of basic types, channel names, or functions.

The input programs provided to the PE are annotated in order to indicate which expressions can be executed at partial evaluation-time. The annotations are computed by the BTA and extend the syntax of terms, yielding a *two-level language* [7,15] described by the full grammar of Equation (3). Underlined expressions are dynamic (of the second stage) while the other are static (of the first stage). `lift` c translates a first order static constant into a dynamic one.

The partial evaluation of a two-level expression e is defined by $\mathcal{P}[\![e]\!]\rho$ where ρ is an environment containing global variables (of first and higher order) shared by all the processes. When the PE, denoted `CPev` finds a free variable in the program being partially evaluated, it looks for its value in ρ, which can be seen as a global memory shared by all the processes and which is used to define all auxiliary functions called by `CPev`.

When `CPev` finds an instruction `fork` e in the program being treated, it creates a new process in which e is applied to a copy `CPev`' of `CPev`. The programs

$$
\begin{aligned}
\mathcal{P}[\![c]\!]\rho &= c\\
\mathcal{P}[\![x]\!]\rho &= \rho(x)\\
\mathcal{P}[\![\texttt{fun}\ x\ \texttt{=>}\ e]\!]\rho &= \lambda v.\mathcal{P}[\![e\{x \leftarrow v\}]\!]\rho\\
\mathcal{P}[\![\texttt{rec}\ f\ x\ \texttt{=>}\ e]\!]\rho &= \lambda v.\mathcal{P}[\![e\{x \leftarrow v\}\{f \leftarrow \texttt{rec}\ f\ x\ \texttt{=>}\ e\}]\!]\rho\\
\mathcal{P}[\![(e_0\ @\ e_1)]\!]\rho &= (\mathcal{P}[\![e_0]\!]\rho)\ (\mathcal{P}[\![e_1]\!]\rho)\\
\mathcal{P}[\![\texttt{if}\ e_0\ e_1\ e_2]\!]\rho &= \textsc{if}(\mathcal{P}[\![e_0]\!]\rho)(\mathcal{P}[\![e_1]\!]\rho)(\mathcal{P}[\![e_2]\!]\rho)\\
\mathcal{P}[\![\texttt{lift}\ c]\!]\rho &= \textsc{build-const}(c)\\
\mathcal{P}[\![\texttt{channel()}]\!]\rho &= \textsc{channel}()\\
\mathcal{P}[\![\texttt{fork}\ e_0]\!]\rho &= \textsc{fork}\ (\mathcal{P}[\![e_0]\!]\rho)\\
\mathcal{P}[\![\texttt{receive}\ e_0]\!]\rho &= \textsc{receive}\ (\mathcal{P}[\![e_0]\!]\rho)\\
\mathcal{P}[\![\texttt{send}\ e_0\ e_1]\!]\rho &= \textsc{send}\ (\mathcal{P}[\![e_0]\!]\rho)\ (\mathcal{P}[\![e_1]\!]\rho)
\end{aligned}
$$

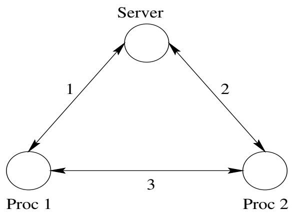
$$
\begin{aligned}
\mathcal{P}[\![\underline{c}]\!]\rho &= \textsc{build-const}(c)\\
\mathcal{P}[\![\underline{\texttt{fun}}\ x\ \texttt{=>}\ e]\!]\rho &= \textsc{let}\ nvar = \textsc{newvar}()\ \textsc{in}\\
&\qquad \textsc{build-fun}(nvar, \mathcal{P}[\![e\{x \leftarrow nvar\}]\!]\rho)\\
\mathcal{P}[\![\underline{\texttt{rec}}\ f\ x\ \texttt{=>}\ e]\!]\rho &= \textsc{build-rec}(f, x, \mathcal{P}[\![e]\!]\rho)\\
\mathcal{P}[\![(e_0\ \underline{@}\ e_1)]\!]\rho &= \textsc{build-app}(\mathcal{P}[\![e_0]\!]\rho, \mathcal{P}[\![e_1]\!]\rho)\\
\mathcal{P}[\![\underline{\texttt{if}}\ e_0\ e_1\ e_2]\!]\rho &= \textsc{build-if}(\mathcal{P}[\![e_0]\!]\rho, \mathcal{P}[\![e_1]\!]\rho, \mathcal{P}[\![e_2]\!]\rho)\\
\mathcal{P}[\![\underline{\texttt{channel()}}]\!]\rho &= \textsc{build-chan}()\\
\mathcal{P}[\![\underline{\texttt{fork}}\ e_0]\!]\rho &= \textsc{build-fork}(\mathcal{P}[\![e_0]\!]\rho)\\
\mathcal{P}[\![\underline{\texttt{receive}}\ e_0]\!]\rho &= \textsc{build-rcv}(\mathcal{P}[\![e_0]\!]\rho)\\
\mathcal{P}[\![\underline{\texttt{send}}\ e_0\ e_1]\!]\rho &= \textsc{build-send}(\mathcal{P}[\![e_0]\!]\rho, \mathcal{P}[\![e_1]\!]\rho)
\end{aligned}
$$

Fig. 1. Evaluation rules of the partial evaluator

`CPev` and `CPev`' are the same and call the same auxiliary functions. On the contrary, when a function `fun` x `=>` e is applied, the effective parameter is directly substituted for x in e and ρ is left unchanged. This approach allows us to avoid the problems related to name clashes in different processes, as well as problems related to the closure of functions which have been communicated.

Note that we only define one kind of variables. In order to ensure that `CPev` behaves correctly when an unknown variable x is found, we extend the environment ρ by $\rho(x) = \llcorner x \lrcorner$, where $\llcorner x \lrcorner$ denotes the piece of code corresponding to the variable x. Doing so, `CPev` builds a residual piece of code for each variable for which the value is unknown at partial evaluation time.

In Figure 1, we describe the behavior of `CPev`. Functions written in small caps correspond to operations of the meta-language which are used to implement `CPev`. Actually, this language is the first-level language corresponding to the first two lines of the grammar of Equation (3). It enables self-application of the PE and makes `CPev` compatible with the Futamura's projections [5], as shown in [12]. The rules used to evaluate sequential expressions are usual, see for instance [8]. $\mathcal{P}[\![\texttt{channel()}]\!]\rho$ creates a new channel name. The evaluation of $\mathcal{P}[\![\texttt{fork}\ e]\!]\rho$ creates a new process which evaluates $\mathcal{P}[\![e]\!]\rho$ in ρ. When a reception $\mathcal{P}[\![\texttt{receive}\ e]\!]\rho$ is found, e is evaluated in ρ, yielding a result α and, next, the communication `receive` α is done. Similarly, for an emission $\llcorner\texttt{send}\ e_0\ e_1\lrcorner$, the expressions e_0 and e_1 are evaluated and the results are used to realize the communication.



```
p_1 : let query = send γ_1 request_1 in        p_2 : let query = send γ_2 request_2 in
      let α_1 = receive γ_1 in                        let α_2 = receive γ_2 in
          send α_1 data                                   receive α_2

                    server : let c_1 = receive γ_1 in
                             let c_2 = receive γ_2 in
                             let α = channel() in
                             let foo = send γ_1 α
                             in send γ_2 α
```

Fig. 2. Creation of a communication link between two processes, via a server

For second-level expressions, the sub-expressions are partially evaluated and a residual term is built. The functions of the form BUILD-FUN are auxiliary functions used to build the residual terms. They are defined in the environment ρ. In the next Section, we show how concurrent programs are partially evaluated by `CPev`.

4 Experimental Results

Partial evaluation of concurrent programs allows one to execute at compile-time the static communications of a distributed application. Here we describe some experiments realized with our implementation of `CPev`.

Our first example is given by the program of Figure 2, in which two processes p_1 and p_2 create a communication link between themselves by consulting a server called s. p_1 and p_2 are linked to the server by channels γ_1 and γ_2. The channel name used for the communications between p_1 and p_2 is provided by the server and is named α_1 in p_1 and α_2 in p_2.

We show how the specialization of this application for a particular network, i.e. in the case where the channels γ_1 and γ_2 are statically known and in which the server is able to know at compile-time the communication link that must be used for the communications between p_1 and p_2. The data exchanged between the processes p_1 and p_2 are assumed to be unknown at partial evaluation time.

We model this application by the program on the next page, in which the communication channels between the server s and the processes p_1 and p_2, as well as the contents of the message, are provided as input parameters.

Since the variable `data` is assumed dynamic, the actual communication between p_1 and p_2 cannot be achieved at partial evaluation-time. This is indicated in the above program by the symbol _ preceeding the related communication primitives. These annotations can be obtained by the BTA introduced in [12].

```
define p1-p2 = fun g1 g2 data ->
                let p1 = fork (let query = send g1 p2Id in
                               let a1 = receive g1 in
                                 _send a1 data)
                in (* p2 *)   (let query = send g2 p1Id in
                               let a2 = receive g2 in
                                 _receive a2) ;
```

We specialize this program in a context where the channels γ_1 and γ_2 are known and where the server is able to compute the communication link to be used for the communications between p_1 and p_2. So, γ_1 and γ_2 are known names and the program is encoded to be understood by the partial evaluator. In addition, we indicate that the variable `data` is unknown. Concurrently with the partial evaluation of the program describing the processes p_1 and p_2, we run the program corresponding to the server in order to allow the execution of the static communications. This corresponds to the following commands (rewritten for a better understanding).

```
> define ch1 = channel() ;
> define ch2 = channel() ;
> define p1-p2-encoded = encode p1-p2 with g1=ch1, g2=ch2, data=? ;
> run {let server = (fork (let c1 = receive ch1 in
                           let c2 = receive ch2 in
                           let p1p2Ch = channel() in
                           let foo1 = send ch1 p1p2Ch
                           in send ch2 p1p2Ch))
       in CPev @ p1-p2-encoded} ;
```

During specialization, the processes p_1 and p_2 are created and the communications with the server are executed. So, the effective channel name used as a communication link between p_1 and p_2, say `#ch`, is inserted in the code of these processes. We obtain the encoding of two residual processes, related to the specialized versions of p_1 and p_2 as shown hereafter.

```
{ send #ch data } | { receive #ch }
```

Our second example is given by the program of Figure 3, which describes a system composed of two processes exchanging a message sliced into packets. We assume the packets have a constant size and that their number depends on the size of the message. The process p_1 first sends to the process p_2 the size of the message (assumed, for the sake of simplicity, to be equal to the number of packets) and, next, builds and sends the packets. The process p_2 receives the size of the message, realizes as many packet receptions as needed, and rebuilds the message. We specialize this program for the particular case in which the size of the message is known at partial evaluation time, but not the contents of the message. For example, this happens in larger systems when the message is defined by a reference on a memory zone declared by a malloc-like primitive. The related annotations of the program are given in Figure 3 a). If we assume

a)

```
p1 : let foo = send γ size
     in (rec f m s →
            if s=1 then
              send (lift γ) (head @ m)
            else
              let foo = send (lift γ)
                               (head @ m)
              in f @ (tail @ m) (s-1)
        ) @ msg size
```

```
p2 : let size = receive γ
     in (rec g s →
            if s=1 then
              receive (lift γ)
            else
              let h= receive (lift γ)
              in append @ h (g @ (s-1))
        ) @ size
```

b)

```
p1 : let symb_1 = send γ (head @ msg)
     in send γ (tail @ msg)
```

```
p2 : let symb_2 = receive γ
     in append @ symb_2 (receive γ)
```

Fig. 3. Slicing of messages into packets of known size and unknown contents. a) Annotated version of the program. b) Residual program obtained by partial evaluation with size= 2

that the size of the message is 2, then Figure 3 b) shows the residual program produced by our partial evaluator. The first communication concerning the size of the message was executed and the loops were unrolled. Our last example is given by the program of Figure 4 a) which describes a system with two processes exchanging a message. When the message is received, a checksum is done. If the test fails, which corresponds to a transmission error, a message is sent to the emitter in order to indicate that the message must be sent again. Otherwise an acknowledgment is sent to the emitter. We specialize this system for a particular network which is assumed to be error free or to handle errors at some lower

a)

```
p1 : rec emitter γ →
        let foo = send (lift γ) data in
        let ack = receive γ in
            if (error @ ack) then
              emitter @ γ
            else
              lift ()
```

```
p2 : rec receiver γ →
        let msg = receive (lift γ) in
        let check = checksum @ msg in
        let foo = send γ check in
            if (error @ check) then
              receiver @ γ
            else
              lift ()
```

b)

```
p1 : let symb_1 = send γ data
     in ()
```

```
p2 : let symb_2 = receive γ
     in ()
```

Fig. 4. Partial evaluation of a communication protocol with error detection. a) Annotated version of the program. b) Residual program obtained assuming that the network is error free

layer of the transfer protocol. Thus, we assume that the `checksum` function always indicates that the received message is correct. As shown is Figure 4 b), we obtain by partial evaluation a new program in which the communications related to the acknowledgments are removed.

5 Conclusion

In this article, we introduced a partial evaluator for programs written in a concurrent functional language and we presented and discussed the results obtained for various small concurrent example programs.

The partial evaluator uses informations about the topology of the communications of the concurrent program. These informations are established using a sophisticated control flow analysis. The quality of the CFA is crucial, since its precision is directly related to the one of the BTA. The CFA we used is described in [13].

Partial evaluation of concurrent systems allows the specialization of applications for particular execution contexts, as shown in the examples of Section 3, i.e. assuming static knowledge of the network topology, or assuming some knowledge on the message sizes or contents. Concerning this latter example, we showed how to specialize a communication protocol by slicing messages into packets. The specialization was for a fixed size of message. This approach was first introduced by Muller *et al.* who used a partial evaluator to specialize the RPC protocol (Remote Procedure Call) w.r.t. the kind of data transmitted [14]. However, due to the fact that the partial evaluator used in [14] only reduced sequential programs, Muller *et al.* could not statically execute some communications and had to use run-time specialization techniques instead [4]. Partial evaluation of the communications, as proposed in this article, allows us to statically realize similar specializations without using any additional techniques or knowledge.

Finally, note that our partial evaluator is compatible with Futamura's projections [5] and automatic compiler generation by self-application of the partial evaluator. The compiler generator related to `Pev` was obtained in [12], where it is described in detail.

References

1. Dave Berry, Robin Milner, and David N. Turner. A Semantics for ML Concurrency Primitives. In *Proceedings of the ACM-SIGPLAN Symposium on Principles of Programming Languages POPL'92*. ACM, 1992. 507
2. Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control Flow Analysis for the Pi-Calculus. In *Concur'98*, number 1466 in Lecture Notes in Computer Science, pages 84–98. Springer Verlag, 1998. 505
3. Anders Bondorf and Jesper Jorgensen. Efficient analyses for realistic off-line partial evaluation: Extended version. DIKU Research Report 93/4, University of Copenhagen, 1993. 504, 506

4. Charles Consel and François Noël. A General Approach for Run-time Specialisation and its Application to C. In *Proceedings of the ACM-SIGPLAN Symposium on Principles of Programming Languages, POPL'96*. ACM, 1996. 512
5. Yoshihito Futamura. Partial Evaluation of Computation Processes - an Approach to a Compiler-compiler. *Systems, Computers, Controls*, 2(5):49–50, 1971. 507, 508, 512
6. Marc Gengler and Matthieu Martel. Self-applicable Partial Evaluation for the Pi-Calculus. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulations, PEPM'97*, pages 36–46. ACM, 1997. 504, 505
7. Marc Gengler and Matthieu Martel. Des étages en Concurrent ML. In *Rencontres Francophones du Parallélisme, Renpar10*, 1998. 507
8. 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. 508
9. Haruo Hosoya, Naoki Kobayashi, and Akinori Yonezawa. Partial Evaluation Scheme for Concurrent Languages and its Correctness. In *Europar'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 625–632. Springer Verlag, 1996. 504, 505
10. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, 1993. 504
11. Mihnea Marinescu and Benjamin Goldberg. Partial Evaluation Techniques for Concurrent Programs. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulations, PEPM'97*, pages 47–62. ACM, 1997. 504, 505
12. Matthieu Martel. *Analyse Statique et Evaluation Partielle de Systèmes de Processus Mobiles*. PhD thesis, Université de la Méditerranée, Marseille, France, 2000. 505, 506, 507, 508, 510, 512
13. Matthieu Martel and Marc Gengler. Communication Topology Analysis for Concurrent Programs. In *SPIN'2000*, volume 1885 of *Lecture Notes in Computer Science*. Springer Verlag, 2000. 505, 506, 512
14. Gilles Muller, Eugen-Nicolae Volanschi, and Renaud Marlet. Scaling up Partial Evaluation for Optimizing the Sun Commercial RPC Protocol. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulations, PEPM'97*, pages 101–111. ACM, 1997. 505, 512
15. Flemming Nielson and Hanne Riis Nielson. *Two-level Functional Languages*. Cambridge University Press, 1992. 507
16. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999. 505
17. John H. Reppy. An operational semantics of first-class synchronous operations. Technical Report TR-91-1232, Department of Computer Science, Cornell University, Ithaca, 1991. 507
18. John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. 507
19. Olin Shivers. *Control Flow Analysis of Higher Order Languages*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1991. Technical Report CMU-CS-91-145. 505
20. Kirsten L. Solberg, Flemming Nielson, and Hanne Riis Nielson. Systematic Realisation of Control Flow Analyses for CML. In *Proceedings of the ACM-SIGPLAN In-*

ternational Conference on Functional Programming, ICFP'97, pages 38–51. ACM, 1997. 505, 506