Partial Evaluation and Distributed Systems

Denis Bechet
CRIN-CNRS & INRIA Lorraine
Campus scientifique, B.P. 239
54506 Vandœuvre-les-Nancy Cedex
France
e-mail: bechet@loria.fr

Abstract

Partial evaluation has been studied mainly for sequential languages. In this paper, we want to discuss of this concept for parallel programs in a distributed environment. We show that standard specializers fail to return good programs and point out the reasons for this failure. Finally, a solution based on symbolic evaluation, type specialization and standard partial evaluation is proposed. It is illustrated by an example of parallel program with a discussion about its specific specialization. This example is a general parallel syntax analyser and it emphasizes perfectly our ideas of what parallelism offers and what partial evaluation could be within this context.

Keywords: Partial evaluation, parallelism, distributed system, syntax analyser, grammar, symbolic evaluation, data specialization.

1 Introduction

This paper describes an attempt to mix parallelism and partial evaluation. This is motivated by the increasing interest for parallelism based on linear logic [Gir87]. The geometry of interaction [Lam90, GAL92, Abr90], interaction nets [Laf90], games semantics [Bla, LS91] are promising theories for understanding concurrency and relation between processes. We have already worked on partial evaluation of interaction nets [Bec92].

However, here, we are interesting to know if classical partial evaluation (i.e. based on binding time analysis [Jør92, Con88, JSS89]) is able to specialize parallel programs, what the problems introduced by this concept are, and what can be done to solve them. From our point of view, concurrency brings problems similar to side-effects since a *send* instruction can be seen as an assignment to a variable and a *receive* instruction as an access to this variable. As a consequence, we can use partial evaluators which take side-effects into account as

in [Con88, Guz88, Har77, LS88, Sch84]. However, for those specializers, side-effects are considered as an additional mechanism to a pure language and they does not try to optimize them. But, since communications are a very important mechanism in parallel programming, those systems are inadequate. For this class of programs, the specializer needs to analyze closely the relations between processes.

In this article, we propose an algorithm for specializing parallel programs. It is split up in three steps. The first one does a symbolic evaluation that try to find the relations between parallel processes. During the second step, programs are modified by specializing the type of the messages exchanged between processes and the instructions involved in communications. The last step consists in a classical specialization of each parallel program. The first step is the important part of the specializer. The two others are more classical.

This paper is organized as follows. After a section which introduces the model of parallelism, the problems of partial evaluation of parallel programs are pointed out. Then, a large example is introduced and its specialization is discussed. Finally, the partial evaluation algorithm is described and a report of its application to the example is given.

2 A model of concurrency and communications

One way to model parallelism consists in introducing new objects, called *tasks*, primitives for communications and a mechanism that creates new tasks. A task is an independent process with its own local variables and communicating through *ports* with other tasks. Ports are grouped two by two (on two different tasks) to form a channel. It is obviously a distributed system with explicit channels (so, broadcasting is not a relevant operation because it means that every task can communicate with all the others). A running system is a network with tasks as nodes and channels as edges. At any moment, it can be characterized by the net of tasks and channels, by the internal state of each task, and by the messages that wait to be read through ports. To implement this idea, the simplest way is to add, to a sequential language, a level of abstraction that defines tasks as a kind of procedure, special instructions for sending and receiving messages and a mechanism that describes how to activate new tasks.

Our language for tasks is an extension of a functional language (SML or CAML [For89]). Channel types are declared using the keyword `channel`. It is a sum of classical types and of channels:

```
<Ch_Decl>      ::= channel <Ch_Name> = <Ch_Case_List>
<Ch_Case_List> ::= <Ch_Case> [ | <Ch_Case_List> ]
<Ch_Case>      ::= <Ch_Case_Id> [ of <Type> ]
```

For instance, a stream of integers can be defined as follows:

```
channel int_stream = 'next_int of int ;;
```

Channels are bi-directional and a task can send and receive through the same port. Ports are independent objects. At any moment a task possesses a list of ports and it can give some of them by sending them to an other task (in this case, the type of the message includes a channel).

Tasks are defined using the special word `task`:

```
<Task> ::= task <Task_Name> <Pattern_List> = <Instruction>
```

The constructions for sending or receiving messages are:

```
<Port_Instruction>::= send  <Port_Expr> to <Port>
                     receive <Port> of <Port_Case_List>
<Port_Expr>       ::= <Ch_Case_Id> [ ( <Expr_List> ) ]
<Port_Case_List>  ::= <Port_Case> [ | <Port_Case_List ]
<Port_Case>       ::= <Ch_Id> [ ( <Var_List> ) ] -> <Instruction>
```

To activate a task, one have to call this task as if it were a procedure.

Below is a task that sends the stream of integers $n..1$ through a port (n is an input integer):

```
task send_int out n =
      if n = 0 then send channel_end to out
               else send 'next_int n to out ;
                    send_int out (n-1) ;;
```

A dual task that receives a stream of integer and prints them is defined by:

```
task print_int_stream inp =
           receive inp of
                channel_end -> void
              | 'next_int n -> print n ; print_int_stream inp ;;
```

The special value `channel_end` indicates that the channel was closed by the sending task.

3 Partial evaluation and tasks

We have seen that tasks are a special kind of procedures. So, partial evaluation can be applied to a task when its arguments are partially instantiated. However, classical partial evaluation fails generally to specialize efficiently a set of tasks because of the essentially dynamic behaviour of messages going through channels. For instance, if we define the task `main` as:

```
task main () = send_int(!p1,10) ; print_int_stream(p1) ;;
```

This task prints the integers 10..1. The problem is that, even if the task `send_int` can be specialized, `print_int_stream` has a dynamic argument and nothing can be done for it. This problem occurs for each channel since what we give to a task is a reference to a port and not its content which is unknown when the task is activated.

In the next sections, we present a more complex example which was the initial motivation of this work. It will serve as a starting point for a discussion on the specialization of parallel programs. Moreover, this example is interesting in itself since it is a syntax analyser and this kind of programs was seldom tackled in the domain of partial evaluation.

4 A significant application: a parallel syntax analyser

In this section, we present a parallel algorithm for syntax analysis. This program takes a grammar and a stream of terminal symbols and returns a syntax tree or, when a syntax error occurs, a stream of syntax trees corresponding to a partial analysis of the stream. The idea behind this algorithm is very simple. In a first step, the input stream of terminals is read and a task is created for each terminal. Each task can communicate with its predecessor and its successor.

After this initial step, each task tries to select the rule which must be applied to its symbol. For this purpose, it exchanges informations with its immediate neighbours and eliminates the rules that not fit. For instance, let us consider the following grammar:

```
E -> T + E | T
T -> F * T | F
F -> ( E ) | id
```

This grammar defines the well-formed arithmetic expressions with addition and multiplication operators. For terminal, the selection of the rule is immediate because they appear only once in the right members of the rules. It is not the case for non-terminals which really have to choose which rule they want to use. This choice is made with the knowledge of its neighbours. For instance, if we consider the series of symbols `T + T * F`, the first `T` knows that it must select the rule `E -> T` because it is at the beginning of the series and its right neighbour is `+`. The second `T` and the symbol `F` must use the rule `T -> T * F`

After this step (in fact, at the same time), when two or more neighbours have decided that they must use the same rule, the tasks merge in a single one. For the above example, the three tasks corresponding to `T * F` merge. Finally, a last operation transforms a complete right member of a rule by the corresponding non-terminal. For the example, the merge task for `T * F` is transformed into the task associated to the single non-terminal `T`. Those steps are then executed once

more. The process terminates when there is only one task (the one associated to the root symbol) or if the system detects a syntax error.

Here is the analysis of the stream `(id+id)*id*id` with the above grammar.

```
I:< ( id + id ) * id * id >
F:[<] [(] [id] [+] [id] [)] [*] [id] [*] [id] [>]
T:[<] [(] F [+] F [)] [*] F [*] F [>]
F:[<] [(] [F] [+] [F] [)] [*] [F] [*] [F] [>]
T:[<] [(] T [+] T [)] [*] [F] [*] T [>]
F:[<] [(] [T] [+] [T] [)] [*] [F] [*] [T] [>]
M:[<] [(] [T] [+] [T] [)] [*] [F,*] [T] [>]
T:[<] [(] [T] [+] E [)] [*] [F,*] [T] [>]
F:[<] [(] [T] [+] [E] [)] [*] [F,*] [T] [>]
M:[<] [(] [T,+] [E] [)] [*] [F,*,T] [>]
T:[<] [(] [T,+] [E] [)] [*] T [>]
F:[<] [(] [T,+] [E] [)] [*] [T] [>]
M:[<] [(] [T,+,E] [)] [*] [T] [>]
T:[<] [(] E [)] [*] [T] [>]
F:[<] [(] [E] [)] [*] [T] [>]
M:[<] [(] [E] [)] [*,T] [>]
M:[<] [(,E,)] [*,T] [>]
T:[<] F [*,T] [>]
F:[<] [F] [*,T] [>]
M:[<] [F,*,T] [>]
T:[<] T [>]
F:[<] [T] [>]
T:[<] E [>]
F:[<] [E] [>]
M:[<,E,>]
T:{End}
S:{End}
```

Each line corresponds to a parallel step when this step modifies at least one of the states of the tasks. The first letter indicates which step is applied: `I` for *initialisation*, `F` for *filtering*, `M` for *merging*, `T` for *transformation* and `S` for *stop*. The symbol associated to the tasks are shown with brackets if it has succeeded in selecting only one rule. For merged tasks the list of symbols is printed, surrounded with brackets. We begin with the list of tasks associated to the input stream `(I+I)*I*I`. After a filtering step, each task has chosen one rule but they can not merge because the rules selected by each task do not correspond each other. However, a transformation step transforms terminals when the right member of the rule has only one symbol. This process continues until it succeeds to derive the axiom.

This syntax algorithm is parallel and distributed. The reductions are made as early as possible in parallel. The decisions are local to a task which does

not know the global state of the system but only the state of its neighbours. The languages that can be analysed by this mechanism depend of the degree of informations exchanged between tasks. The LL class of languages corresponds to the knowledge of the symbols associated to the two neighbours whereas exchanging the list of items gives the power of the LR class. But, because the process is purely symmetric, the classes are larger than the classic ones (if a language can be recognised then the reverse one is also recognised).

5 Partial evaluation and syntax analysers

Obviously, we can specialize this algorithm when the grammar is given. It is well-known that general syntax analysers are not very efficient. But, we can build by hand programs which analyse very quickly a sentence for a particular grammar (in fact, programs like Yacc [Joh] does this transformation for us). So, this domain is interesting when someone wants to test a specializer because he knows that it already exists good solutions. Moreover, general analysers are often complex and give a good test in a realistic environment.

For what we know on this subject, this problem was seldom considered in the literature [Dyb85, Pag90, EO82]. It is a little strange because the most popular example of the potential power of partial evaluation is the automatic generation of a compiler from an interpreter and because the first operation of an interpreter is the transformation of a sentence into an abstract representation. One of the specifications of an interpreter is the grammar associated to its language. This situation is certainly due to the complexity of this problem and as [Dyb85] says in conclusion, that partial evaluator requires some improvements before being used in this domain.

If we analyse the failure of the specialization of a general analyser using an exiting algorithm, we see that, even if the grammar is instantiated, the analyser uses very often the conjunction of the input list of terminals and the grammar in branching conditionals. The static data are never used alone but nearly always with an internal state and the next symbol of the input stream. So, since the input is dynamic, each sub-expression can not be reduced and is annotated as dynamic.

To solve this problem, one has to make a case analysis on the input and with this partial information tries to evaluate the program until it needs more informations from the input. After this step, we obtain a new program that can quickly analyse the first element of information. But, it is not sure that this version will be able to continue an efficient analysis of the rest of the input. So, we have to introduce a new case statement for the second element and so on. This transformation is the way a program is transformed into a finite automaton and it can be seen as a attempt to creat the argument/result graph of a function. Of course, it is not sane because an arbitrary function can not be described by a finite graph. It is the case for the problem of syntax analysis (though for a

lexical analyser, it is a possible solution).

Thus, this algorithm is not feasible in a standard way. But, the situation is not the same for parallel programs. To see this, we must look at the reasons to introduce parallelism. A first argument says that if the system can compute different expressions at the same time, the time to obtain the result will obviously decrease when compared with a sequential mechanism. There is an other argument for the parallel point of view which is that tasks are not functions. They interact each other without knowing the global state but with the satisfaction that the job they do is a small brick to achieve the global aim. Thus, we can assume that their internal state is finite and the messages they send are also finite (i.e. the infinity is only the result of the infinity of task networks we can create). For instance, the algorithm for the above parallel syntax analysis has this property. For this class of parallel programs, we can use case analysis.

6 The specialization algorithm

To implement the above idea, we have divided the work into three main steps. The first one try to record messages that can be sent through ports and the different activations of tasks. Then, the channel types are specialized and the `send` and `receive` instructions in the bodies of the tasks are modified. Finally, the tasks are specialized. In fact, one can use an already existing specializer (here for functional programming), modifying it for the new set of instructions. Here, we will mainly talk about the first two steps. To simplify the problem, we assume that parallel instructions appear only in the body of tasks. The classical functions are supposed to be purely functional (with no side-effect, no `send` nor `receive` instruction and no call to a task).

1. For the first step, the recording of the messages which are sended through a type of channel and of the different calls to tasks is done by an abstract evaluation of the different tasks. This work is done incrementaly since generating new messages and new calls can create new ones. The termination of this algorithm is implicitly included in the assumption that there are only a finite number of messages exchanged between tasks and a finite number of calls to a task.

   The main loop takes a list of channel case identifiers (the constructor of messages) with the list of messages already associated to it. It has also a list of task activations. For each task call, this information is an initial environment which bounds each variable of the list of arguments to a value. A dynamic argument is associated to the symbolic value `'dynamic`. Moreover, the symbolic value `'port` is given to port expressions (even if it is dynamic). Then, the body of each task is evaluated for one of the recorded call.

The symbolic evaluator takes an expression, a local environment and global definitions (i.e. for global values like functions).

- For standard instructions, the evaluator has the same behaviour as an evaluator for this language except that if a variable which is dynamic appears in an expression, then this expression is considered as dynamic. When a variable has to be added to the environment, the expression which defining it is evaluated. If this expression is a port, the value `'port` is associated to this variable otherwise its value is the result of the evaluation of this expression. For a branching instruction, if the condition is static, the evaluator continues with the selected branch. Otherwise, it examines each branch. A call to a function is executed if and only if all the arguments are known.
- For a `send` instruction, the evaluator tries to reduce the message which will be sended. If it succeed in computing the message and if this message was not generating by the previous steps then it records this message which will be examined by the next loop. If the expression is dynamic then it records that it fails to find a static message for this channel constructor. It means that when a task wants to read this kind of message, there must be a general case to handle an unknown value.
- For a `receive` instruction, each case is evaluated as if the task has received one of the messages associated to this channel constructor. So, there are as many evaluations as there are messages for this constructor.
- Finally, for the activation of a task (i.e. a call to a task), the arguments are evaluated and the call is recorded if it is new.

After executing this process for each task call, the list of new messages of the previous loop is appended to the previous ones and another loop is executed. The algorithms stops when no new message nor new call is generated.

2. This first step gives for each channel constructor, a list of messages and casualty the value "dynamic". With this information, the channel types can be specialized. For each constructor, we add to the channel declaration as many constructors as there are messages. The original constructor is kept in the declaration if the value "dynamic" is also in the list. Then the body of the tasks are modified to take into account this modification. This is done by transforming all the `send` and `receive` instructions which appear in the body of tasks.

   - For a `send` instruction, it means that this instruction is transformed into a case on the value of the message which is compared with the

values of the corresponding channel constructor and if it matches, rather than sending this value, we send the new channel constructor which corresponds to this value. An "else" case is included if the value "dynamic" is also a possibility. The message that will be sended is the original one.

- For a `read` instruction, cases are added for new constructors. The right parts of those cases are a copy of the general right part of the corresponding constructor. However, a binding of the variables of the pattern is appended and their values are the value of the message associated to the new constructor.

3. The last operation is to create as many specialized versions of a task as there are calls to this task. For each call, a new task name is generated. The declaration of arguments contains only dynamic variables and the body is partially evaluated. Static expressions are reduced, out of reach branches are erased, a call to a task is replaced by a call to the specialize version: it is a classical partial evaluation process.

7 Partial evaluation of the syntax analyser

The result of the partial evaluation of the syntax analyser when the grammar is known is good when compared to programs obtained with a tool like Yacc. Initially, the specializer starts with the main task, with its grammar argument instantiated, and with the list of terminals associated to the input stream (the possible values of this stream). Then, in a first step, the symbolic evaluator finds the messages and the tasks associated to a terminal. After that, and progressively, non-terminal are treated until no more symbol remains. The new program is composed of a set of specialized versions of the filtering task for each set of items that are reachable and a set of merging tasks for each subpart of the right member of the rules. The transformation task has completely disappeared since it does not need to communicate to others tasks. Each task contains only `send` and `receive` instructions and calls to other tasks. The variables have disappeared except for the one bound to ports. The number of tasks corresponds approximatively to the number of states which are necessary for an LR analysis. In fact, each task is a small automaton depending on two inputs (the states of its neighbours).

8 Conclusion and perspectives

This work was motivated by the interest for parallelism based on linear concepts. The model of small processes linked two by two and interacting each other is surely a promising domain. Here, this philosophy was used to convince us

that partial evaluation needs other mechanisms than a binding time analysis. Because processes interact each other rather than taking an input, computing and returning a value, specialization must focus its attention on the relations between tasks. It is the main aim of the first step of the above algorithm. However, a lot of work remains to improve and generalize the ideas behind this system. We are currently working on a translation of functional programs into a parallel abstract machine (interaction nets [Laf90]) and we want to introduce partial evaluation at this level rather than a the level of the initial language because we think that the above algorithm can be implemented more efficiently in this formalism than on functional programs [Bec92].

References

[Abr90] S. Abramsky. Computational interpretations of linear logic. Imperial College Research Report DOC 90/20, Departement of Computing, Imperial College of Science, Technology and Medecine, London, UK, 1990.

[ASU86] A. Aho, R. Sethi, and J. Ullman. Syntax analysis. In *Compilers: Principles, techniques and tools*, chapter 4. Addison-Wesley Publishing Company, Inc., Reading, Mass, 1986.

[Bec92] D. Bechet. Partial evaluation of interaction nets. In M. Billaud et al., editors, *WSA '92, Static Analysis, Bordeaux, France, September 1992. Bigre vols. 81-82, 1992*, pages 331–338. IRISA, 1992.

[Bla] A. Blass. A game semantics for linear logic.

[Con88] C. Consel. New insights into partial evaluation: The Schism experiment. In H. Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 1988. (Lecture Notes in Computer Science, vol. 300)*, pages 236–246. Springer-Verlag, 1988.

[Dyb85] H. Dybkjær. Parsers and partial evaluation: An experiment. Student Project 85-7-15, DIKU, University of Copenhagen, Denmark, July 1985. 128 pages.

[EO82] A.P. Ershov and B.N. Ostrovsky. Systematic construction of a program for solution of a particular problem from a certain class examplified by syntactic analyzers. *Doklady Akademii Nauk SSSR*, 266(4):803–806, 1982. (In Russian).

[For89] Projet Formel. *The CAML Reference Manual.* INRIA-ENS, March 1989. Version 2.6.

[GAL92] G. Gonthier, M. Abadi, and J.J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the Nineteenth ACM Symposium om Principles of Programming Languages*, pages 15–26. ACM, 1992.

[Gir87] J.Y. Girard. Linear logic. In *Theoretical Computer Science*, volume 50, pages 1–102. 1987.

[Guz88] M.A. Guzowski. Towards developing a reflexive partial evaluator for an interesting subset of Lisp. Master's thesis, Dept. of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, January 1988.

[Har77] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation.* PhD thesis, Linköping University, Sweden, 1977. Linköping Studies in Science and Technology Dissertations 14.

[Joh] S.C. Johnson. *YACC : Yet Another Compiler-Compiler.* Bell Labs. Unix Programmer's Manual.

[Jør92] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992*, pages 258–268. ACM, 1992.

[JSS89] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[Laf90] Y. Lafont. Interaction nets. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 95–108. ACM, January 1990.

[Lam90] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 16–30. ACM, January 1990.

[LS88] G. Levi and G. Sardu. Partial evaluation of metaprograms in a multiple worlds logic language. *New Generation Computing*, 6(2,3):227–247, 1988.

[LS91] Y. Lafont and T. Streicher. Game semantics for linear logic. In *Sixth IEEE Symposium on Logic in Computer Science.* IEEE, 1991.

[Pag90] F.G. Pagan. Comparative efficiency of general and residual parsers. *Sigplan Notices*, 25(4):59–65, April 1990.

[Sch84] R. Schooler. Partial evaluation as a means of language extensibility. Master's thesis, 84 pages, MIT/LCS/TR-324, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1984.