Tutorial Notes on Partial Evaluation

Charles Consel

Pacific Software Research Center
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology*
consel@cse.ogi.edu

Olivier Danvy

Department of Computing and Info. Sciences
Kansas State University †
danvy@cis.ksu.edu

Abstract

The last years have witnessed a flurry of new results in the area of partial evaluation. These tutorial notes survey the field and present a critical assessment of the state of the art.

1 Introduction

Partial evaluation is a source-to-source program transformation technique for specializing programs with respect to parts of their input. In essence, partial evaluation removes layers of interpretation. In the most general sense, an interpreter can be defined as a program whose control flow is determined by its input data. As Abelson points out, [43, Foreword], even programs that are not themselves interpreters have important interpreter-like pieces. These pieces contain both compile-time and run-time constructs. Partial evaluation identifies and eliminates the compile-time constructs.

1.1 A complete example

We consider a function producing formatted text. Such functions exist in most programming languages (*e.g.*, `format` in Lisp and `printf` in C). Figure 1 displays a formatting function written in Scheme [21]. Given a channel, a control string, and a list of values, this function outputs text in the channel by interpreting the control string to determine how to format the list of values.

For conciseness, this function handles only three formatting directives: `~N`, `~S` and `~%`. The first two directives specify that the corresponding element in the list of values must be printed as a number or as a string, respectively. The third directive is interpreted as an end-of-line character. Any other character is printed verbatim. For simplicity, we assume that the control string matches the list of values.

Most of the time, `format` is called with a constant control string. This situation is ideal for partial evaluation, particularly when `format` is called with the same constant control string repeatedly. Specializing `format` with respect to this control string enables one to interpret it only once. The specialized function takes a channel and a list of values and returns an updated channel. It is built as a dedicated combination of printing operations. Figure 2 presents a version of `format` specialized with respect to the control string `"~N is not ~S~%"`.

The interpretive overhead of `format` has been entirely removed. All the operations manipulating the control string have been performed at compile-time. No references to the control string are left in the residual program. The specialized function only consists of operations manipulating the run-time arguments, *i.e.*, the channel and the list of values to be formatted.

This example illustrates the essential purpose of partial evaluation: eliminating interpretive overhead — here the interpretation of the control string.

1.2 Applications

Because of its conceptual simplicity, partial evaluation has been applied to a wide variety of areas that include compiling and compiler generation [31, 34, 39, 58, 68, 70, 73], string and pattern matching [28, 40, 69, 92, 102], computer graphics [83], numerical computation [8], circuit simulation [5], and hard real-time systems [89].

Partial evaluation has been studied in the context of a wide variety of programming languages. In the area of logic programming, partial deduction is the focus of much work [44, 47, 77, 76]. Partial evaluators for imperative languages like Pascal [82] and C [3] have been developed. In equational languages, partial evaluation have been used to optimize rewriting techniques [101], and more recently, it has been applied to Lafont's interaction nets [6].

A broader discussion and more detailed references on contemporary work in the area of partial evaluation and on applications of partial evaluation can be found in Jones, Gomard, and Sestoft's new book [66].

Overview. This paper is organized as follows. Section 2 presents the principles and practice of partial evaluation. Section 3 briefly reviews the state of the art about partial evaluators for Scheme and imperative languages. Section 4 addresses the problem of termination. Section 5 analyzes the tradeoff between space and time when using partial evaluation. Section 6 presents some applications. Section 7 addresses related work. Finally, Appendix A discusses self-application.

2 Principles and Practice of Partial Evaluation

This section first describes the extensional aspects of partial evaluation. Then, we present strategies to achieve partial evaluation and outlines work formalizing these strategies. Finally we discuss generalized forms of partial evaluation.

*19600 N.W. von Neumann Drive, Beaverton, Oregon 97006-1999, USA.

†Manhattan, Kansas 66506, USA. Part of this work was supported by NSF under grant CCR-9102625.

To appear in the proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages, January 1993, Charleston, South Carolina. [Revision, October 28, 1992]

```
;;; format: Port * ControlString * List(Value) -> Port
(define format
  (lambda (port control-string values)
    (let ([end (string-length control-string)])
      (letrec ([traverse
                (lambda (port offset values)
                  (if (= offset end)
                      port
                      (case (string-ref control-string offset)
                        [(#\~) (case (string-ref control-string (+ offset 1))
                                 [(#\N) (traverse (write-number port (car values))
                                                  (+ offset 2)
                                                  (cdr values))]
                                 [(#\S) (traverse (write-string port (car values))
                                                  (+ offset 2)
                                                  (cdr values))]
                                 [(#\%) (traverse (write-newline port)
                                                  (+ offset 2)
                                                  values)]
                                 [else (error 'format "illegal control string: ~S" control-string)])]
                        [else (let ([new-offset
                                     (letrec ([upto-tilde
                                               (lambda (new-offset)
                                                 (cond
                                                   [(= new-offset end) new-offset]
                                                   [(equal? (string-ref control-string new-offset) #\~) new-offset]
                                                   [else (upto-tilde (+ new-offset 1))]))])
                                       (upto-tilde offset))])
                                (traverse (write-string port (substring control-string offset new-offset))
                                          new-offset
                                          values))])))])
        (traverse port 0 values)))))

;;; given write-string:   Port * String -> Port
;;;       write-number:   Port * Number -> Port
;;;       write-newline:  Port          -> Port
```

Figure 1: A formatting function

```
;;; for any port p and list of two values vs,
;;; (format.1 p vs) = (format p "~N is not ~S~%" vs)

(define format.1
  (lambda (port values)
    (write-newline
      (write-string
        (write-string
          (write-number port (car values))
          " is not ")
        (car (cdr values))))))
```

Figure 2: Specialized version of Figure 1

2.1 Partial evaluation: what

Given a general program and part of its input, we want to specialize this program with respect to this known information.

Consider a program p and its input i, and say that somehow we can split i into a *static* (*i.e.* known) part s and a *dynamic* (*i.e.* unknown) part d. For example, p might take two arguments, one of which is a static value, and the other one of which is a dynamic value. Given a specializing function $\mathcal{S}$, we can specialize p with respect to s:

$$\mathcal{S}(p, \langle s, \_\rangle) \;=\; p_s$$

By definition, running the residual program p_s must yield the same result as the general program would yield, provided both terminate:

$$\text{run}\; p\; \langle s,\, d\rangle \;=\; \text{run}\; p_s\; \langle \_,\, d\rangle$$

As illustrated in the introduction, p_s can run faster than p.

In fact, $\mathcal{S}$ is Kleene's S^m_n-function [74]. This function is computable and thus it can be implemented: the result is what is called a partial evaluator (hereafter denoted *PE*).

$$\text{run}\; PE\; \langle p,\, \langle s,\, \_\rangle\rangle \;=\; \mathcal{S}(p, \langle s, \_\rangle)$$

2.2 Partial evaluation: how

Specializing a program with respect to all of its input amounts to running this program and producing a constant residual program, *i.e.*, a program that takes an empty input and produces an already computed value. Therefore, a partial evaluator must include an interpreter to perform reductions at specialization-time.

Dually, specializing a program with respect to none of its input amounts to produce a (possibly simplified) version of this program. Therefore, a partial evaluator must include a compiler to construct the residual program. In the common case where the source and the target languages coincide this compiler essentially mimics the identity function. Otherwise, it is a simple translator [60].

When specializing a program with respect to some known parts of its input, a partial evaluator corresponds to a non-standard interpreter: it evaluates the expressions depending on static data (*i.e.*, data available at specialization-time) and it reproduces the expressions depending on dynamic data (*i.e.*, data available only at run-time). A partial evaluator propagates constant values and folds constant expressions. It also inlines functions by unfolding calls, and produces specialized functions by residualizing calls. A *monovariant* specializer produces at most one specialized function for every source function. A *polyvariant* specializer can produce many specialized versions of a source function [18].

2.3 Online *vs.* offline partial evaluation

Contemporary partial evaluators are divided in two classes: *online*, monolithic partial evaluators and *offline*, staged partial evaluators. Before Jones's Mix system, all partial evaluators were online [7, 55, 79, 99]. Today, both online and offline partial-evaluation strategies are the subject of active research.

An online partial-evaluator is a non-standard interpreter. The treatment of each expression is determined on the fly. Online partial-evaluators in general are very accurate but at the price of a considerable interpretive overhead.

Most programming languages are not simply implemented with an interpreter. Instead, their implementation is structured with a compiler and a run-time system. Implementing partial evaluation makes no exception, and offline partial evaluators are structured with a preprocessing phase $\overline{PE}$ and a processing phase $\underline{PE}$. The preprocessing phase $\overline{PE}$ usually includes a *binding-time analysis* [68, 88]. Given the binding-time signature of a source program (*i.e.*, which part of the input is static and which part is dynamic), the binding-time analyser propagates this information in the source program, determining for each expression whether it can be evaluated at compile-time or whether it must be evaluated at run-time. This binding-time information is then used to guide the processing phase $\underline{PE}$ that performs the specialization proper. Binding-time analysis is necessarily approximate[1] and thus offline partial-evaluators are usually less accurate than online ones.

Binding-time analysis has been intensively studied in the framework of abstract interpretation [13, 25, 32, 35, 87] as well as in the framework of type theory [52, 59, 90, 98, 103]. In offline, binding-time based partial-evaluators, accurate binding-time information is critical because it determines the degree of the actual specialization. Existing binding-time analyses handle higher-order functions and data structures. Newer ones are polyvariant.

The binding-time information determines whether an expression can be evaluated at partial-evaluation time or must be evaluated at run-time. In an earlier work, we proposed to stage *PE* further by shifting the interpretation of binding-times from $\underline{PE}$ to $\overline{PE}$ [24, 29]. This shift substantially simplifies the specializer.

To get the best of both worlds, online and offline partial-evaluation can be combined as follows [11, 107]. Whenever the exact binding-time property of an expression can be determined, offline partial evaluation is used. Otherwise, the treatment for this expression is postponed until specialization-time, when concrete values are available. Bondorf's partial evaluator for term-rewriting systems uses this strategy [11].

[1] For example, a conditional expression whose consequent and alternative branches are not bound at the same time is classified to have the latest binding-time of its components.

2.4 Formalizing partial evaluation

Much effort is devoted to specifying and proving offline partial-evaluation. Gomard defines the denotational semantics of the specialization process for the λ-calculus and proves its correctness [53]. Launchbury formulates a binding-time analysis for a first-order applicative language with projections [62, 78]. This line of work aims at proving the correctness of offline partial-evaluation with respect to the standard semantics of the source language. More generally, Consel and Khoo formally define online and offline partial-evaluation for a first-order applicative language [35]. They relate the standard semantics of the language to an online partial-evaluation semantics. Then they show that binding-time analysis is an abstraction of the online partial-evaluation semantics. Finally, they derive the specialization semantics from the binding-time analysis and the online partial evaluation semantics. The method can be applied to other languages.

2.5 Generalizing specialization

In the traditional presentation of partial evaluation, "specializing a program with respect to part of its input" is implicitly understood as "given the actual value for some formal parameters of a program, construct a specialized program with fewer formal parameters". This point of view has been progressively refined over the years. For example, Launchbury's partial evaluator is based on projections [78]. The input of a source program is divided into a static projection and a dynamic projection.

In their parameterized partial-evaluation, Consel and Khoo allow a program to be specialized not only with actual values of its input, but also with respect to any property of this input [32]. The generalization applies for both online and offline strategies. Parameterized partial-evaluation has been implemented at CMU [22] and at Yale [72] for a first-order subset of ML. Both systems handle partially-static data. The latter implementation includes both online and offline partial-evaluation phases.

2.6 The structure of a source program

In general, partial evaluation forces one to be very conscious about the structure and properties of one's source programs. Typically, a program "specializes well" when it processes the static part of the input independently of the dynamic part. For example, consider the following two functions:

```
(lambda (x y z)
  (+ (+ x y) z))
```

```
(lambda (x y z)
  (+ x (+ y z)))
```

Unless the partial evaluator is instructed that addition is associative, specializing these two functions with respect to their two first parameters (say, 2 and 3, respectively) does not produce the same result:

```
(lambda (z)
  (+ 5 z))
```

```
(lambda (z)
  (+ 2 (+ 3 z)))
```

The first function "specializes better" than the second one because it processes the static part of the input (*i.e.*, `x` and `y`) independently of the dynamic part of the input (*i.e.*, `z`).

Thus some information about the binding-time signature of a source program (*i.e.*, which part of the input is static and which part is dynamic) enables one to structure this program to make it specialize better.

2.7 The structure of a residual program

The residual program obtained by specializing a source program with respect to some static data is structured like the static data.

For example, the control string `"~N is not ~S~%"` is built iteratively with a formatting directive for numbers, then a few characters, then a formatting directive for strings, and then a newline. Correspondingly, the residual program of Figure 2 is built to iteratively output a number, then a constant string, then a string, and then a newline.

This observation makes it easier to read residual programs. In addition, occurrences of some static data in the residual program signify that, in the source program, the static data are not processed independently of the dynamic data. This leads to strategies for restructuring source programs to "improve their binding-times" [14, 15, 30].

Independently, one is often surprised to find redundant tests in one's residual programs — usually a tell-tale of unexpected redundancies in the source programs. As such, a partial evaluator is a useful programming tool.

3 State of the Art

We first review the state of the art of partial evaluators for call-by-value functional languages such as Scheme [21], and then of partial evaluators for imperative languages.

3.1 Applicative languages

Weise's partial evaluator Fuse is an online system and aims at applying partial evaluation to practical problems, such as circuit simulation [107]. The design of Fuse is distinct from earlier online partial evaluators in that it uses graphs as an intermediate language and has a strategy for increasing sharing in residual programs [71, 94, 95].

Bondorf's partial evaluator Similix is an offline polyvariant system and was explicitly designed to be self-applicable and mostly automatic, based on a fixed specialization strategy [16]. The system handles recursive equations, customizable primitive operators and global side-effects. It also includes a binding-time debugger [86]. Since then, it has been extended to handle higher-order functions [13] and more recently partially-static values [15]. Today, the new version of Similix is based on Henglein's efficient type-inference for binding-time analysis [59]. Similix is freely available and is used as a black box in Harnett and Montenyohl's investigation of programming languages [58]. Independently, Gengler and Rytz have extended the system with a polyvariant binding-time analysis and with partially static values [48, 96].

Consel's partial evaluator Schism is an offline polyvariant system with a flexible specialization strategy, higher-order functions, and partially-static values [27]. Both the binding-time analysis and the specialization are polyvariant. The system includes a binding-time based programming-environment [36]. Both source programs and specialized programs are expressed in Scheme, extended with ML-like datatypes.

3.2 Imperative languages

Partial evaluation of imperative programs has received much attention recently. Meyer developed an online partial evaluator for a subset of Pascal [82]. Nirkhe and Pugh [89] describe a partial evaluator for hard real-time problems where programs are constrained by the user to keep a tight control over the transformation process. Andersen reports a self-applicable partial evaluator for a subset of the C programming language where binding-time annotations are supplied by the user [3].

Partial evaluation of imperative programs is difficult because of the lack of referential transparency. The program transformation phase must take into account the notion of state and thus is more complicated than in a functional setting [54]. Unless imperative features are encapsulated in some language constructs and so side-effects are disciplined, it is difficult to reason about the flow of static data. Duplication of side-effects and aliasing are the main concerns [1].

4 Termination

Due to its basic strategy — unfolding calls and specializing functions, partial evaluation can loop in two ways: either by unfolding infinitely many function calls or by creating infinitely many specialized functions. Both problems can be avoided naively by limiting the number of unfolded calls and the number of specialized functions, but often this strategy appears unsatisfactory. In practice, some programs require much unfolding while they are traversing static data and some others require many residualizations. In this section, we review the strategies used in Mix, Similix, Schism, and Fuse.

In the Mix system, the problem was first treated by inserting annotations by hand in the source program, to indicate which call should be unfolded and which should be residualized [67]. Later on, static analyses were devised to annotate first-order programs automatically [100].

The use of binding-time analysis enables one to insert more accurate annotations. The strategy adopted in Similix, for example, is very simple and appears to be applicable in most situations in an automatic way: dynamic conditional-expressions (*i.e.*, conditional expressions whose test do not evaluate to a static value) are selected as specialization points and all procedure calls are unfolded [16].

Schism offers a more flexible annotation strategy: filters, that can be used both in an online or in an offline strategy [23, 24, 27]. The user can equip each function with a filter, specifying under which conditions a call to this function should be unfolded and, if the call needs to be residualized, which parameters should be used for the specialization of this function. Writing one's own filters provides the user with full control over specialization. Filters can also be generated automatically, based on any strategy. For example, an analysis corresponding to the strategy of Similix is available in Schism [27].

As an online partial-evaluator, Fuse keeps a dynamic cache of program points [107, Section 3]. Specialization naturally terminates when all loops that are unrolled statically terminate or are broken by a cache hit. Otherwise, an arbitrary bound is needed.

In general, the treatment of function calls not only determines the termination of the specialization process, but also has an impact on the size and efficiency of the residual program.

5 Data *vs.* Code

Using partial evaluation is based on a tradeoff: taking more space for programs and data may produce faster computations while taking less space for programs and data may produce slower computations.

5.1 Xphoon

The manual pages documenting Poskanzer and Leres' program xphoon illustrate program specialization at its best. Xphoon displays a picture of the moon and was created by compiling the program loading a full-screen bitmap together with a bitmap representing the moon: experience shows that the specialized program is both faster than loading a full-screen bitmap and smaller than the bitmap file representing the moon. So in this case, program specialization wins both spacewise and timewise.

5.2 Pitfalls

The two basic strategies of partial evaluation — unfolding and specialization — can of course lead to unsatisfactory results, even if specialization terminates. For example, a program may be overly specialized and contain many instances of the same piece of code, just differing with a single constant. At the other end of the spectrum, if partial evaluation is too conservative, the residual program may contain many occurrences where further specialization actually would pay off.

Care must also be taken when unfolding function calls to avoid duplicating computations. This can lead to the specialization of a linear program into an exponential one [100]. This problem is met in C with the following macro.

```
#define inlinedplus(x) = x + x;
```

Applying `inlinedplus`, for example, to a function call can make a linear-time looking code run in exponential time.

6 Some Concrete Applications

This section illustrates how partial evaluation can be used to derive non-trivial programs. In particular, this method stresses the fact that non-trivial programs are often instances of simpler ones [97, Chapter 5].

6.1 Pattern Matching

Let us consider the following string matching problem: does a string occur within a text? A variety of solutions have been proposed — for example, by Knuth, Morris, & Pratt [75] and by Boyer & Moore [17] — that essentially solve this problem in linear time with respect to the size of the string and of the text. It is now folklore in the partial-evaluation community how to derive the Knuth, Morris & Pratt method out of the naive, quadratic program: After a (partial) match, we know that a piece of the dynamic input string is identical to (a prefix of) the static pattern. This means that we can now match the pattern against a shifted copy of itself, rather than the input string; and the outcome of such a match can be decided at specialization-time [28].

Rather than rewriting the source program to make it keep a static track of dynamic values, one can also obtain this residual program by generalizing the partial evaluator [46, 57, 102]. For example, let us specialize the following function by letting its first parameter be 10:

```
(lambda (s d)
  (case d
    [(1) (+ s d)]
    [(2) (- s d)]
    [else d]))
```

A naive strategy would yield the following residual program.

```
(lambda (d)
  (case d
    [(1) (+ 10 d)]
    [(2) (- 10 d)]
    [else d]))
```

However, a strategy that keeps a static track of dynamic values across conditional expressions can do a better job and produce the following residual program.

```
(lambda (d)
  (case d
    [(1) 11]
    [(2) 8]
    [else d]))
```

This simple step is enough to produce residual programs that traverse the dynamic data linearly [40, 69]. Regarding string matching, any traversal of the string and the text leads to a specialized program that is linear over the dynamic string. Thus if the traversal goes from left to right and the string is static, the residual program mimics the effect of the Knuth, Morris, & Pratt string-matching algorithm. If the traversal goes from left to right and the text is static, the residual program is structured like a Weiner tree [106] (named a "position tree" by Aho, Hopcroft, and Ullman [2]). If the traversal goes from right to left and the string is static, the residual program mimics the effect of the Boyer & Moore string-matching algorithm. In fact, we have observed that any traversal of the static data leads to a linear residual program[2] (which is remarkable considering how much work has been invested to prove the expected linearity of some string-matching algorithms). Hansen has identified several variations around the Boyer & Moore string matching algorithm [56]. More recently, Queinnec and Geffroy have identified several other traversals of the static data corresponding to other well-known matching algorithms [92].

But even though partial evaluation can be used to generate linear-time programs, there is no guarantee about the size of these programs, nor about the time taken by the partial evaluator to produce them. In particular, it would be surprising that the partial evaluator generates a program mimicking the effect of Knuth, Morris, & Pratt in time linear to the static string — whereas Knuth, Morris, & Pratt's algorithm first constructs a "failure table" in time linear to the string, and then traverses the text in linear time.

So partial evaluation does offer a safe way to remove the interpretive overhead of pattern matching and to construct linear-time residual programs. However, other insights are necessary to produce small programs quickly.

6.2 Partial evaluation applied to operating systems

An important trend in operating system development is the restructuring of the traditional monolithic operating system kernel into independent servers running on top of a minimal/micro kernel [50]. This approach results in modular and flexible operating systems. Also operating systems can be written in high-level programming languages like Scheme [64]. However, these advantages come at a price: microkernel-based modular operating systems do not provide performance comparable to monolithic ones.

With the Synthesis kernel, Pu and his group have shown that microkernel-based operating systems can be optimized by generating specialized kernel routines [91]. Their work demonstrated that efficiency can be obtained without compromise on modularity and flexibility.

Although the Synthesis kernel has been a breakthrough in microkernel-based operating systems, it is based on an ad-hoc specialization process and requires the code to be specialized to be manually annotated. Such a process is tedious and error-prone.

Since microkernel-based operating systems can now be written in high-level programming languages there is no reason why partial evaluation cannot be used to perform the kind of specializations performed in the Synthesis kernel.

In [37], Consel, Pu and Walpole describe a research project aimed at using partial evaluation to derive automatically implementations of operating system components from generic specifications. They outline the necessary extensions to partial evaluation required for this derivation.

[2] We even proved this property, based on a partial evaluator like Similix that is guaranteed not to duplicate residual expressions.

7 Related Work

All optimizing compilers include constant propagation and folding [1]. The need for optimizing compilers to be efficient has motivated Wegman and Zadeck to study this subject on its own [105]. For another example, Deutsch's interactive program verifier, like many other theorem provers, includes a simplification phase performing static reductions [41]. Mosses's compiler generator SIS includes a phase for compiler-generation time reductions and allows for compile-time reductions [85]. Appel's technique of "reopening closures" explicitly aims at specializing functions at compile-time [4]. In fact, Lombardi and Raphael's main tool in their pioneer work on incremental computation was partial evaluation [79].

The investigations above have one point in common: they use program transformation as a phase in a larger system. Therefore this phase needs to be efficient. Alternatively, transforming a program can be the main goal of a system, and then the emphasis is put first on understanding what is going on as a preliminary step to making the transformation efficient [19].

We also observe a new trend in using one of the main techniques of partial evaluation — polyvariance — in modern compilers [20, 38].

Acknowledgements

We are grateful to Anindya Banerjee, Andrzej Filinski, John Hatcliff, Jim Hook, Julia Lawall, Jürgen Koslowski, Karoline Malmkjær, Tim Sheard, and Erik Ruf for commenting earlier versions of these notes on short notice.

References

[1] A. D. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] L. O. Andersen. Self-applicable C program specialization. In Consel [26], pages 54–61.

[4] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[5] W. Au and D. Weise. Automatic generation of compiler simulation through program specialization. In *IEEE Conference on Design Automation*, pages 205–210, 1991.

[6] Denis Bechet. Partial evaluation of interaction nets. In WSA'92 [109], pages 331–338.

[7] L. Beckman, A. Haraldsson, O. Oskarsson, and E. Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7(4):319–357, 1976.

[8] A. Berlin. Partial evaluation applied to numerical computation. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, 1990.

[9] D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[10] C. Böhm. Subduing self-application. In *16th International Colloquium on Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[11] A. Bondorf. Towards a self-applicable partial evaluator for term rewriting systems. In Bjørner et al. [9].

[12] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, University of Copenhagen, DIKU, Copenhagen, Denmark, 1990. DIKU Report 90-17.

[13] A. Bondorf. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[14] A. Bondorf. Similix manual, system version 3.0. Technical Report 91/9, Computer Science Department, University of Copenhagen, 1991.

[15] A. Bondorf. Improving binding times without explicit CPS-conversion. In *ACM Conference on Lisp and Functional Programming*, pages 1–10, 1992.

[16] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[17] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):62–72, 1976.

[18] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[19] R. M. Burstall and J. Darlington. A transformational system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.

[20] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, SIGPLAN Notices, Vol. 24, No 7, pages 146–160, 1989.

[21] W. Clinger and J. Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[22] C. Colby and P. Lee. An implementation of parameterized partial evaluation. In WSA'91 [108], pages 82–89.

[23] C. Consel. New insights into partial evaluation: the Schism experiment. In *ESOP'88, 2^{nd} European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 236–246. Springer-Verlag, 1988.

[24] C. Consel. *Analyse de Programmes, Evaluation Partielle et Génération de Compilateurs.* PhD thesis, Université de Paris VI, Paris, France, June 1989.

[25] C. Consel. Binding time analysis for higher order untyped functional languages. In *ACM Conference on Lisp and Functional Programming*, pages 264–272, 1990.

[26] C. Consel, editor. *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* Research Report 909, Department of Computer Science, Yale University, 1992.

[27] C. Consel. Report on Schism'92. Research report, Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, USA, 1992.

[28] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.

[29] C. Consel and O. Danvy. From interpreting to compiling binding times. In *ESOP'90, 3^{rd} European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 88–105. Springer-Verlag, 1990.

[30] C. Consel and O. Danvy. For a better support of static data flow. In Hughes [63], pages 496–519.

[31] C. Consel and O. Danvy. Static and dynamic semantics processing. In *ACM Symposium on Principles of Programming Languages*, pages 14–23, 1991.

[32] C. Consel and S. C. Khoo. Parameterized partial evaluation. Research Report 865, Yale University, New Haven, Connecticut, USA, 1991. To appear in *Transactions on Programming Languages and Systems*. Extended version of [33].

[33] C. Consel and S. C. Khoo. Parameterized partial evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 92–106, 1991.

[34] C. Consel and S. C. Khoo. Semantics-directed generation of a Prolog compiler. In *3^{rd} International Symposium on Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 135–146. Springer-Verlag, 1991.

[35] C. Consel and S.C. Khoo. On-line & off-line partial evaluation: Semantic specifications and correctness proofs. Research Report 896, Yale University, New Haven, Connecticut, USA, 1992.

[36] C. Consel and S. Pai. A programming environment for binding-time based partial evaluators. In Consel [26], pages 62–66.

[37] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. Research report, Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, USA, 1992.

[38] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Fourth IEEE International Conference on Computer Languages*, pages 96–105, 1992.

[39] Pierre Crégut. *Machines à environnement pour la réduction symbolique et l'évaluation partielle*. PhD thesis, Université Paris VII, 1991.

[40] O. Danvy. Semantics-directed compilation of non-linear patterns. *Information Processing Letters*, 37:315–322, March 1991.

[41] L. P. Deutsch. An interactive program verifier. Technical Report CSL-73-1, Xerox PARC, May 1973.

[42] A. P. Ershov, D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldsson, and W. L. Scherlis, editors. *Selected Papers from the Workshop on Partial Evaluation and Mixed Computation*, volume 6 (2,3) of *New Generation Computing*. OHMSHA. LTD. and Springer-Verlag, 1988.

[43] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1991.

[44] D. A. Fuller and S. Abramsky. Mixed computation of Prolog. In Bjørner et al. [9].

[45] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls 2, 5*, pages 45–50, 1971.

[46] Y. Futamura and K. Nogi. Generalized partial computation. In Bjørner et al. [9].

[47] J. Gallager and M. Codish. Specialisation of Prolog and FCP programs using abstract interpretation. In Bjørner et al. [9].

[48] M. Gengler and B. Rytz. A polyvariant binding time analysis handling partially known values. In WSA'92 [109], pages 322–330.

[49] R. Glück. Towards multiple self-application. In Hudak and Jones [61], pages 309–320.

[50] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the USENIX Summer Conference*, 1990.

[51] C. K. Gomard. Higher order partial evaluation – HOPE for the lambda calculus. Master's thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, 1989.

[52] C. K. Gomard. Partial type inference for untyped functional programs. In *ACM Conference on Lisp and Functional Programming*, 1990.

[53] C. K. Gomard. A self-applicable partial evaluator for the lambda-calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

[54] C. K. Gomard and N.D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12:123–144, 1991.

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

[56] T. A. Hansen. Transforming a naive pattern matcher into efficient pattern matchers. Technical report, DAIMI, 1991.

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

[58] S. Harnett and M. Montenyohl. Towards efficient compilation of a dynamic object-oriented language. In Consel [26], pages 82–89.

[59] F. Henglein. Efficient type inference for higher-order binding-time analysis. In Hughes [63], pages 448–472.

[60] C. K. Holst. Language triplets: The AMIX approach. In Bjørner et al. [9], pages 167–185.

[61] P. Hudak and N. D. Jones, editors. *Partial Evaluation and Semantics based Program Manipulation*. Vol. 26, No 9. ACM SIGPLAN Notices, 1991.

[62] J. Hughes. Backward analysis of functional programs. In [42], pages 187–208, 1988.

[63] John Hughes, editor. *FPCA'91, 5^{th} International Conference on Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, 1991.

[64] S. Jagannathan and J. Philbin. A foundation for an efficient multi-threaded Scheme system. In *ACM Conference on Lisp and Functional Programming*, pages 345–357, 1992.

[65] N. D. Jones. Partial evaluation, self-application and types. In *17th International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 639–659. Springer-Verlag, 1990.

[66] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993. To appear.

[67] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[68] 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.

[69] J. Jørgensen. Generating a pattern matching compiler by partial evaluation. In Simon L. Peyton Jones, Guy Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990*, pages 177–195. Springer-Verlag, 1991.

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

[71] M. Katz and D. Weise. Towards a new perspective on partial evaluation. In Consel [26], pages 29–37.

[72] S. C. Khoo. *Parameterized Partial Evaluation: Theory and Practice*. PhD thesis, Yale University, 1992. Forthcoming.

[73] S. C. Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In Hudak and Jones [61], pages 211–222.

[74] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[75] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM*, 6(2):323–350, 1977.

[76] H. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *ACM Symposium on Principles of Programming Languages*, 1982.

[77] A. Lakhotia and L. Sterling. ProMiX: A Prolog partial evaluation system. In L. Sterling, editor, *The Practice of Prolog*, chapter 5, pages 137–179. MIT Press, 1991.

[78] J. Launchbury. *Projection Factorisation in Partial Evaluation*. PhD thesis, Department of Computing Science, University of Glasgow, Scotland, 1990.

[79] L. A. Lombardi and B. Raphael. Lisp as the language for an incremental computer. In E. C. Berkeley and D. G. Bobrow, editors, *The Programming Language Lisp: Its Operation and Applications*, pages 204–219. MIT Press, Cambridge, Massachusetts, 1964.

[80] K. Malmkjær. On static properties of specialized programs. In WSA'91 [108], pages 234–241.

[81] K. Malmkjær. Predicting properties of residual programs. In Consel [26], pages 8–13.

[82] U. Meyer. Techniques for partial evaluation of imperative languages. In Hudak and Jones [61], pages 94–105.

[83] T. Mogensen. The application of partial evaluation to ray-tracing. Master's thesis, University of Copenhagen, DIKU, Copenhagen, Denmark, 1986.

[84] T. Mogensen. *Binding Time Aspects of Partial Evaluation*. PhD thesis, University of Copenhagen, DIKU, Copenhagen, Denmark, 1989.

[85] P. Mosses. *SIS – Semantics Implementation System, reference manual and user guide*. University of Aarhus, Aarhus, Denmark, 1979. Version 1.0.

[86] C. Mossin. Similix binding time debugger manual. Technical report, University of Copenhagen, Copenhagen, Denmark, 1991.

[87] F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

[88] H. R. Nielson and F. Nielson. Automatic binding time analysis for a typed λ-calculus. In *ACM Symposium on Principles of Programming Languages*, pages 98–106, 1988.

[89] V. Nirkhe and W. Pugh. Partial evaluation of high-level imperative program languages with applications in hard real-time systems. In *ACM Symposium on Principles of Programming Languages*, pages 269–280, 1992.

[90] J. Palsberg and M. I. Schwartzbach. Binding time analysis: Abstract interpretation versus type inference. Technical report, DAIMI, 1992.

[91] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *ACM Computing Systems*, 1(1):11–32, 1988.

[92] C. Queinnec and J. M. Geffroy. Partial evaluation applied to pattern matching with intelligent backtracking. In WSA'92 [109], pages 109–117.

[93] E. Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Department of Computer Science, Stanford University, 1993. (in preparation).

[94] E. Ruf and D. Weise. Using types to avoid redundant specialization. In Hudak and Jones [61], pages 321–333.

[95] E. Ruf and D. Weise. Improving the accuracy of higher-order specialization using control flow analysis. In Consel [26], pages 67–74.

[96] B. Rytz and M. Gengler. A polyvariant binding time analysis. In Consel [26], pages 21–28.

[97] W. L. Scherlis. *Expression Procedures and Program Derivation*. PhD thesis, Department of Computer Science, Stanford University, 1980. Report No. STAN-CS-80-818.

[98] D. A. Schmidt. Static properties of partial evaluation. In Bjørner et al. [9], pages 465–483.

[99] R. Schooler. Partial evaluation as a means of language extensibility. Master's thesis, M.I.T. (LCS), Massachusetts, U.S.A, 1984. TR-324.

[100] P. Sestoft. Automatic call unfolding in a partial evaluator. In Bjørner et al. [9].

[101] D. Sherman, R. Strandh, and I. Durand. Optimization of equational programs using partial evaluation. In Hudak and Jones [61], pages 72–82.

[102] D. A. Smith. Partial evaluation of pattern matching in CLP domains. In Hudak and Jones [61], pages 62–71.

[103] K. L. Solberg, H. R. Nielson, and F. Nielson. Inference systems for binding time analysis. In WSA'92 [109], pages 247–254.

[104] R. S. Sundaresh and P. Hudak. Incremental computation via partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 1–13, 1991.

[105] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 3(2):181–210, 1991.

[106] P. Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[107] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In Hughes [63], pages 165–191.

[108] *Workshop on Static Analysis*, volume 74 of *Bigre Journal*. IRISA, Rennes, France, 1991.

[109] *Workshop on Static Analysis*, volume 81-82 of *Bigre Journal*. IRISA, Rennes, France, 1992.

A Optimizing Partial Evaluation: Self-Application

A.1 Self-application: what

Often, we need to specialize a program p with respect to many different values. Then we are in the situation of running PE several times on p. But PE is just another program that we need to run several times when a part of its input (p) does not change.

Partial evaluation tells us that a faster way to do this is first to produce a version of PE specialized with p (removing the interpretive overhead of PE) and then to run this version instead:

$$\text{run } PE\ \langle PE,\ \langle p,\ \_\rangle\rangle \ = \ PE_p$$

By definition of program specialization, running the residual program on s should yield the same result as PE would yield:

$$\text{run } PE\ \langle p,\ \langle s,\ \_\rangle\rangle \ = \ \text{run } PE_p\ \langle s,\ \_\rangle$$

Because PE_p is obtained by specializing the specializer, this bootstrapping process is called "self-application".[3]

For example, we might need to specialize the program `format` of the introduction with respect to many different control strings s_0, s_1, ... Instead of running the general-purpose partial-evaluator on the same program `format`, we can build a specializer dedicated to `format` (*i.e.*, a program that only knows how to specialize `format`) first, and then run it as many times as needed on the different values:

$$\begin{aligned} \text{run } PE\ \langle PE,\ \langle \texttt{format},\ \_\rangle\rangle &= PE_{\texttt{format}} \\ \text{run } PE_{\texttt{format}}\ \langle \_,\ s_0,\ \_\rangle &= \texttt{format}_{s_0} \\ \text{run } PE_{\texttt{format}}\ \langle \_,\ s_1,\ \_\rangle &= \texttt{format}_{s_1} \end{aligned}$$

[3] The term "self-application" is confusing because the partial evaluator simply processes a copy of itself, just as there are C compilers written in C. So in particular there is no danger of paradox or untypability as in the λ-calculus [10, 65].

In practice, $PE_{\texttt{format}}$ is quite small. Most of this program coincides with what a programmer would write by hand. The rest includes more specific features of the specializer.

Partial evaluation can improve performance even further. Suppose that several programs need to be specialized repeatedly with respect to many different static values. We already know how to optimize the specialization of one program p with many static values. This optimization requires us to specialize PE several times with respect to many different programs p. In other terms, we need to run a program (PE) several times when a part of its input (PE) does not change, which is clearly inefficient. The remedy to this inefficiency is of course to specialize PE with respect to PE first, and then to use the residual program PE_{PE} to transform a source program p into a specializer PE_p dedicated to specializing p — without interpretive overhead.

Since these optimizations are independent of the actual program p, they are applicable to any kind of source programs. For example, when the source program p is the interpreter for a programming language, these optimizations are known as the "Futamura projections" [45]. In this case, PE_p has the functionality of a compiler and PE_{PE} of a compiler generator. Sundaresh and Hudak point out that when p is an incremental interpreter, then PE_p has the functionality of an "incrementalizer" [104].

Self-application is an elegant idea in principle, and was at the basic motivation for Jones's Mix project [68] and for much of the work carried out at DIKU [12, 51, 84]. For example, Malmkjær analyzes dedicated specializers PE_p to predict generic properties of specialized versions of p [80, 81].

A.2 Self-application: how

In a partial evaluator, the static part of the input is the source program and the dynamic part of the input is the input to the source program. To make the partial evaluator "specialize better" (*i.e.*, to make it more suitable for specialization), one should ensure that the source program is processed as independently as possible of its input.

During their PhD studies, Consel, Ruf, and Glück have structured a partial evaluator along these lines to improve self-application [23, 49, 93]. This work aims at obtaining dedicated specializers that are both reasonably small and reasonably effective.

In his Mix project [68], Jones proposed a Gordian solution and stages PE explicitly into a static phase $\overline{PE}$ and a dynamic phase $\underline{PE}$. This staging leads one to reformulate partial evaluation as follows:

$$\text{run } PE\ \langle p,\ \langle s,\ \_\rangle\rangle \ = \ \text{run } \underline{PE}\ \langle(\text{run } \overline{PE}\ p),\ \langle s,\ \_\rangle\rangle$$

In particular, self-application is reformulated as follows:

$$\begin{aligned} \text{run } PE\ \langle PE,\ \langle p,\ \_\rangle\rangle \ = \ \text{run } \underline{PE}\ \langle&(\text{run } \overline{PE}\ \underline{PE}), \\ &\langle(\text{run } \overline{PE}\ p),\ \_\rangle\rangle \end{aligned}$$

In practice, $\overline{PE}$ takes a second parameter, usually the binding-time signature of the first parameter.

Jones's staging effectively eliminates the problem of self-application by ensuring that dedicated specializers are only built out of the run-time part of PE (since they are specialized instances of $\underline{PE}$) and that source programs are processed statically (since they are passed to $\overline{PE}$ beforehand).