Automatic Program Specialization by Partial Evaluation: an Introduction

Robert Glück and Neil D. Jones

DIKU, Department of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen, Denmark
e-mail: {glueck,neil}@diku.dk

Summary: Partial evaluation is an automatic program optimization technique, similar in concept to, but in several ways different from optimizing compilers. Optimization is achieved by changing the times at which computations are performed. A partial evaluator can be used to overcome losses in performance that are due to highly parameterized, modular software. This has a quite remarkable impact on software development because it allows the design of general and reusable software without the penalty of being too inefficient. This papers gives an introduction to automatic program specialization by off-line partial evaluation.

1 Introduction

The past several years have seen a worldwide surge of interest in automatic program transformation and analysis. This is not accidental: the introduction of the computer was a giant step in the *execution* of programs, but the *creation* of software was not directly affected. This activity is still performed by the human ("programming"). While the complexity and volume of software has been growing rapidly, the development of software remains in essence a handcraft that requires enormous amounts of human effort and investment. Even though one often hears terms such as "programming methodology," "best practice" *etc.*, such approaches are only ways of better *organizing* the handcraft.

Partial evaluation is a program optimization technique that can help to *automate* part of the software development process by providing means to tailor generic and highly parameterized software to specific needs and applications.

2 Program Specialization

The problem. Software developers face a serious dilemma when they have

- a class of similar problems
- each of which must be solved efficiently.

On the one hand they want to write general and well-structured programs that are easy to debug, maintain and document. On the other hand programs that solve particular subproblems are often significantly faster, but are time consuming to write and much harder to maintain if all subproblems must be solved well—a small change in the initial specification may require all programs to be modified by hand! It would be ideal if one could get the best of both worlds: write a few highly parameterized and perhaps inefficient programs, and then obtain automatically as many customized and more efficient versions as required.

A solution. A partial evaluator is a tool for program optimization, similar in concept to, but in several ways different from highly optimizing compilers. Optimization is achieved by changing the *times* at which computations are performed. A partial evaluator can be used to overcome losses in performance that are due to highly parameterized software.

Assume that `Pgm` is a program with two inputs `X` and `Y`. Let ⟦`Pgm`⟧ `X Y` denote the application of program `Pgm` to `X` and `Y`. Computation of `Pgm` in one stage is described by

$$\texttt{Out} = [\![\texttt{Pgm}]\!]\ \texttt{X}\ \texttt{Y}$$

Program `Pgm` can be transformed into a specialized version $\texttt{Pgm}_\texttt{X}$ by *precomputing* those parts of `Pgm` that depend only on the input `X`. A partial evaluator `Pe` is a program that takes `Pgm` and `X` as input and produces a specialized version $\texttt{Pgm}_\texttt{X}$ as output. Computation in two stages using a partial evaluator is described by

$$\texttt{Pgm}_\texttt{X} = [\![\texttt{Pe}]\!]\ \texttt{Pgm}\ \texttt{X}$$
$$\texttt{Out} = [\![\texttt{Pgm}_\texttt{X}]\!]\ \texttt{Y}$$

The result of specializing the *source program* `Pgm` with respect to `X` (the *static* input) is a *residual program* $\texttt{Pgm}_\texttt{X}$ that returns the same result when applied to the remaining input `Y` (the *dynamic* input) as the source program `Pgm` when applied to the input `X` and `Y`. All residual programs are faithful to the original program, but are often significantly faster. The main task of a partial evaluator is to recognize which of `Pgm`'s computations can be precomputed at specialization time and which must be delayed until run time.

When is it beneficial? A wide spectrum of apparently problem-specific optimizations can be seen as instances of partial evaluation. Partial evaluation can help in the following frequently occurring situation when parameters vary with different rates:

- a program Pgm is to be executed for many different input pairs (X, Y),
- X is changed less frequently than Y, and
- a significant part of Pgm's computation depends only on X.

As an example consider *ray tracing*, a method for computer graphics. The method is known to give good picture rendition, but is relatively slow because it repeatedly computes information about the ways millions of rays traverse a given scene from different origins and in different directions. Specialization of a ray tracer with respect to a fixed scene transforms the general algorithm into a specialized ray tracer that computes pictures significantly faster than the original, general algorithm.

Speedups. The main motivation for doing partial evaluation is speed: program $\mathtt{Pgm}_\mathtt{X}$ is often faster than Pgm. To describe this more precisely, let $t_{\mathtt{Pgm}}(\mathtt{X},\mathtt{Y})$ be the time to compute $[\![\mathtt{Pgm}]\!]$ X Y. Define the *speedup function* as

$$speedup_{\mathtt{X}}(\mathtt{Y}) = \frac{t_{\mathtt{Pgm}}(\mathtt{X},\mathtt{Y})}{t_{\mathtt{Pgm}_\mathtt{X}}(\mathtt{Y})}$$

Specialization is advantageous if Y changes more frequently than X. To exploit this, each time X changes construct a new specialized $\mathtt{Pgm}_\mathtt{X}$ and run it on various Y until X changes again. Sometimes it may be worthwhile to specialize a program, even though the residual program is executed only once, *i.e.* the time to specialize plus the time to run the residual program is less than running the original program: $t_{\mathtt{Pe}}(\mathtt{Pgm},\mathtt{X}) + t_{\mathtt{Pgm}_\mathtt{X}}(\mathtt{Y}) < t_{\mathtt{Pgm}}(\mathtt{X},\mathtt{Y})$. An analogy, sometimes it is faster to compile a source program and to run it than to interpret it.

Broader perspectives. The following summarizes a number of uses of automatic program specialization, and the possible advantages and improvements that can be achieved.

A general tool for software development. Many program speedups achieved by handwork are specializations—and can be automated. This is not at all a new idea. Quite often, program run times are improved by hand rewriting, a task that is predictably error-prone. The novelty is that program specialization is a *general*, *automatic*, and *application-independent tool*, and is therefore a more *powerful* and *reliable technology* for software development. A potential application is the removal of abstraction layers and interface code present in modern software libraries.

Automated reuse of software. An automatic specialization tool has a quite remarkable impact on software development because it allows the design of generic and reusable software without the penalty of being too inefficient. This involves a different view of specialization: given a program `Pgm`, first (by hand) generalize it to obtain a generic program `Gen`, *e.g.* by adding parameters to give it a *broader functionality*. Then program `Gen` can be reused for different applications by automatically specializing it with respect to specific parameter settings.

Problems of interpretive nature. It has become clear that program specialization is very well suited to problems involving interpretive definitions. Often it is desirable to devise a language for *user-oriented specifications*, to describe parameters in a way more related to the problems being solved than to the software solving them. Examples are numerous (another virtue of specialization!) and include computer simulation and modeling, the implementation of logical meta-systems, and automatic compiling from interpreters.

3 Off-Line Partial Evaluation

A partial evaluator can be seen as a mixture of an interpreter and a compiler: those parts of a program that depend only on static input are executed at specialization time, while code is generated for the rest. Partial evaluation comes in two flavors, *online* and *offline*. The difference between the two lies in the time when the decision whether to generate code or to evaluate an expression is taken. In online partial evaluation this is done "on the fly" at specialization time, while in offline partial evaluation the decision is taken before specialization time. Online techniques sometimes give better specialization, while offline techniques are better for giving feedback to the user. In the remainder of this section we give an introduction to offline partial evaluation.

How is partial evaluation done? In *offline* partial evaluation the transformation process is guided by a *binding-time analysis* performed prior to the specialization phase. The result of the binding-time analysis is a program in which all expressions are *annotated* as either static or dynamic. Their interpretation is simple. Operations annotated as

- *static* are performed at specialization time
- *dynamic* are delayed until run time (*i.e.* residual code is generated).

Example. Consider a program `power` computing the function x^n. The source program in Fortran is shown to the left, the specialized version to the right—assuming that `x` is dynamic and `n` is static (= 9). Dynamic expressions are annotated by underlining, *e.g.* `x = x*x`, while all other constructs are interpreted as static. Computations depending only on the static `n` can be executed at specialization time; the actions depending on the dynamic `x` are deferred until run time. The annotation can be done automatically with a binding-time analysis.

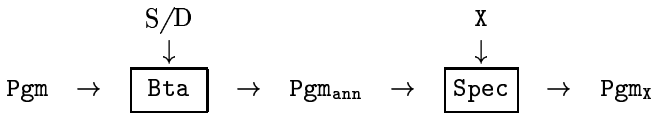
```
c Compute x to the n'th
  INTEGER n, x, power
  ...
  power = 1
1 IF (n .GT. 0) THEN
2    IF (MOD(n,2) .EQ. 0) THEN
        n = n/2
        x = x*x
        GOTO 2
     END IF
     n = n-1
     power = power*x
     GOTO 1
  END IF
  ...
```

```
c Specialized wrt n=9
  INTEGER x, power
  ...
  power = 1
  power = power*x
  x = x*x
  x = x*x
  x = x*x
  power = power*x
  ...
```

The specialized program can be optimized further (*e.g.* by propagating `power = 1`), but this will in any case be done by an optimizing compiler. On the other hand, since compilers lack static input and binding-time analysis, it is unreasonable to expect a compiler to execute static statements and to produce specialized programs. Any attempt to specialize even slightly larger programs by hand without a partial evaluator is already tedious and error prone.

Partial evaluation, like highly optimizing compilers, is no panacea: it may not be worthwhile in all cases. For example knowing the value of `x` (with `n` dynamic) will not significantly aid computing `power`.

How to obtain program annotations? The task of the binding-time analysis is to compute a *division* B of all variables V in a program, given an initial classification of the input variables as either static (S) or dynamic (D). Variables classified as static by the analysis depend only on static input variables; variables classified as dynamic may depend on dynamic input variables. This information is then used to annotate the expressions in a program.



Algorithm. (Monovariant Binding-Time Analysis) *Call the program variables* $V_1, \ldots, V_N$ *and assume that the input variables are* $V_1, \ldots, V_n$, *where* $1 \leq n \leq N$. *Assume that the binding-times* $\overline{b}_1, \ldots, \overline{b}_n$ *for the input variables are given, where* $\overline{b}_i$ *is either* S *(static) or* D *(dynamic). The task is to compute a division for all program variables:* $B = (b_1, \ldots, b_N)$ *which satisfies* $\overline{b}_i = \mathrm{D} \Rightarrow b_i = \mathrm{D}$ *for the input variables. The analysis is done by the following algorithm:*

1. *Construct the initial division* $\overline{B} = (\overline{b}_1, \ldots, \overline{b}_n, \mathrm{S}, \ldots, \mathrm{S})$ *and set* $B = \overline{B}$.
2. *If the program contains an assignment* $V_k \leftarrow \exp$ *where the variable* V_j *appears in* exp *and* $b_j = \mathrm{D}$ *then set* $b_k = \mathrm{D}$ *in* B.
3. *Repeat step 2 until* B *does not change any longer. Then the algorithm terminates with* congruent *division* B *which can be used for specialization.*

Specializing dynamic basic blocks. The general idea is to specialize the control-flow and each particular statement to the static values. Static transitions can be done at specialization time, *e.g.* an `IF` with a static test; dynamic control-flow must be deferred to run time, *e.g.* an `IF` with a dynamic test.

Recall that a basic block is a sequence of statements such that there is a distinguished entry point (the first statement), and the last statement is a control statement (`CALL`, `RETURN`, `IF`, `GOTO`). A *dynamic basic block* (DBB) is a set of basic blocks $B_0 ... B_n$ such that there is

1. a dynamic transition to B_0 (or it the program's first basic block);
2. all transitions among $B_0 ... B_n$ are static;
3. all control statements in the leaf basic blocks are dynamic.

All reachable DBBs are specialized with respect to static values computed at specialization time (the static storage). A done- and a pending-list are used to keep track of already specialized DBBs and those remaining to be specialized. Each time a dynamic transition is reached the done-list is searched whether the 'target' DBB was already specialized with respect to the same static storage. If so, the corresponding jump label is inserted. Otherwise, the DBB has to be specialized with respect to the static storage. This method is known as *poly-variant specialization* because the same DBB may be specialized with respect to different static storage (for more details see *e.g.* [JGS93, And94]).

Effects and limitations. The specialization effects are typically due to *unfolding* (unrolling) of loops and *elimination* of conditionals, *interprocedural constant propagation* (as opposed to intraprocedural), *procedure specialization* (cloning), *precomputation* of *indices*, *coefficients*, and parts of library functions. Experiments show that specialized programs often enable further compiler optimizations (*e.g.* when array indices become known); thus, the choice of the compiler optimization level affects the speedup. The gain in efficiency has its price

(sometimes an increase in program size) and it is not always desirable to fully unfold loops since the number of iterations may be extremely large.

Certain programming styles complicate the analysis and specialization of programs. A partial evaluator user may have to think about a program from a new perspective: what are the binding-time properties of a program? Does one need semantics preserving transformations, called *binding-time improvements*, that make it easier for the partial evaluator to make more operations static?

Historical notes. The idea of obtaining a one-argument function by "freezing" an argument of a two-argument function is classical mathematics. The possibility, in principle, of partial evaluation is contained in Kleene's *s-m-n* Theorem from 1936, an important building block of recursive function theory. On the other hand, efficiency matters were quite irrelevant to these investigations.

The idea to use partial evaluation as a *programming tool* can be traced back to work beginning in the late 1960's. After a period of independent insights in the seventies, the last decade has seen substantial advances both in theory and practice of partial evaluation, as documented by a series of international meetings, *e.g.* [BEJ88, DGT96, PEC, PEW].

Despite the successful application of partial evaluation to declarative languages, such as Lisp or Prolog, only few attempts have been made to exploit partial evaluation in the context of imperative languages. Ershov and his group were the first who investigated imperative languages and mixed computation [Ers78]. Partial evaluators for C and for a Fortran subset have been developed [And94, KKZG95]. The application of partial evaluation to software maintenance has been investigated in [BF94] and an approach to run-time specialization in [CN96]. The book [JGS93] describes approaches to and systems for partial evaluation, and includes a comprehensive bibliography. An electronic version of the bibliography is available from

`ftp://ftp.diku.dk/diku/semantics/partial-evaluation/partial-eval.bib.Z`

Another source of information:

`http://www.diku.dk/research-groups/topps/Research.html`

4 Conclusion

Partial evaluation has been the subject of a rapidly increasing research activity over the past decade since it provides a uniform paradigm for a broad spectrum of work on automatic program manipulation and program generation. It opens the possibility for implementing highly parameterized, modular software without loss of efficiency. Results in different application areas clearly indicate that existing partial evaluation technology is strong enough to improve the efficiency of a

large class of practically oriented programs. Partial evaluation as a tool for "real world" software development remains still a challenging problem. It now appears feasible to transfer this technology to a realistic context.

Bibliography

[And94] Lars Ole Andersen. Program analysis and specialization for the C programming language. DIKU Report 94/19, Department of Computer Science, University of Copenhagen, 1994.

[BEJ88] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[BF94] Sandrine Blazy and Philippe Facon. Partial evaluation for the understanding of Fortran programs. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):535–559, 1994.

[CN96] Charles Consel and François Nöel. A general approach for run-time specialization and its application to C. In *Twenty Third Symposium on Principles of Programming Languages*, pages 145–156, ACM Press, 1996.

[DGT96] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation. Proceedings.* Lecture Notes in Computer Science, Vol. 1110, Springer-Verlag, 1996.

[Ers78] Andrei P. Ershov. On the essence of compilation. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[KKZG95] Paul Kleinrubatscher, Albert Kriegshaber, Robert Zöchling, and Robert Glück. Fortran program specialization. *SIGPLAN Notices*, 30(4):61–70, 1995.

[PEC] *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, 1991, 1993, 1995.

[PEW] *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* 1992, 1994.