Partial Evaluation and the Generation of Program Generators

Neil D. Jones

DIKU, Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
E-mail: neil@diku.dk

1 Introduction

Partial evaluation has been the subject of rapidly increasing activity over the past decade since it provides a unifying paradigm for a broad spectrum of work in program optimization, compiling, interpretation and the generation of automatic program generators [7,14,25].

It is a program optimization technique, perhaps better called program specialization, closely related to but different from Jørring and Scherlis’ staging transformations [27]. It emphasizes, in comparison with [11,27] and other program transformation work, full automation and the generation of program generators as well as transforming single programs.

Much partial evaluation work to date has concerned automatic compiler generation from an interpretive definition of a programming language, but it also has important applications to scientific computing, logic programming, metaprogramming, and expert systems; some pointers are given later.

1.1 Partial Evaluation = Program Specialization

A partial evaluator is given a subject program together with part of its input data, in1. Its effect is to construct a new program Pin1 which, when given p’s remaining input in2, will yield the same result that p would have produced given both inputs. In other words a partial evaluator is a program specializer. In Figure 1 the partial evaluator is called mix.¹

¹Notation: data values are in ovals, and programs are in boxes. The specialized program Pin1 is first considered as data and then considered as code, whence it is enclosed in both. Further, single arrows indicate program input data, and double arrows indicate outputs. Thus mix has two inputs while Pin1 has only one; and Pin1 is the output of mix.
Figure 1: A Partial Evaluator

Figure 2: Specialization of a Program to Compute $x^n$.

Intuitively, specialization is done by performing those of $p$’s calculations that depend only on $\text{in1}$, and by generating code for those calculations that depend on the as yet unavailable input $\text{in2}$. Figure 2 shows a two-input program to compute $x^n$, and a faster program $p_5$ resulting from specialization to $n = 5$.

The technique is to precompute all expressions involving $n$, to unfold the recursive calls to function $f$, and to reduce $x\cdot 1$ to $x$. This optimization was possible because the program’s control is completely determined by $n$. If on the other hand $x = 5$ but $n$ is unknown, specialization gives no significant speedup.

A partial evaluator performs a mixture of execution and code generation actions — surely why Ershov called the process “mixed computation” [14], hence the name $\text{mix}$. 
An Equational Description

Programs are both input to and output from other programs. We will discuss several languages and so assume given a fixed set $D$ of first order data values including all program texts. A suitable choice of $D$ is the set of Lisp’s “list” data as defined by $D = \text{LispAtom} + D^*$, e.g. $(1\ (2\ 3)\ 4)$ is a list of three elements, whose second element is also a list.

An example Lisp-like program is $p =$

\[
\begin{align*}
\text{(define (length x)} \\
\text{ (case x of)} \\
\text{ (()) : 0} \\
\text{(x1 . xrest) : (add 1 (length xrest)))}
\end{align*}
\]

We use the typewriter font for programs and for their input and output. If $p$ is a program in language $L$, then $\llbracket p \rrbracket_L$ denotes its meaning — typically an input-output function.

The subscript $L$ indicates how $p$ is to be interpreted. When only one language is being discussed we often omit the subscript so $\llbracket p \rrbracket_L = \llbracket p \rrbracket$. Standard languages used in the remainder of this article:

$L =$ implementation language
$S =$ source language
$T =$ target language

Letting $V = D + (D \to V)$, the program meaning function $\llbracket \cdot \rrbracket_L$ is of type $D \to V$. Thus

\[
\text{output} = \llbracket p \rrbracket_L \text{ in}_1 \text{ in}_2 \ldots \text{ in}_n
\]

results from running $p$ on input values $\text{in}_1$, $\text{in}_2$, ..., $\text{in}_n$, where $n \geq 0$ (and output is undefined if $p$ goes into an infinite loop).

An Equational Definition

The essential property of a partial evaluator $\text{mix}$ is now formulated more precisely. Suppose $p$ is a source program, $\text{in}_1$ is the data known at stage one (static), and $\text{in}_2$ is data known at stage two (dynamic). Then computation in one stage is described by

\[
\text{out} = \llbracket p \rrbracket \text{ in}_1 \text{ in}_2
\]

Computation in two stages using specializer $\text{mix}$ is described by

\[
\begin{align*}
\text{pin}_1 &= \llbracket \text{mix} \rrbracket p \text{ in}_1 \\
\text{out} &= \llbracket \text{pin}_1 \rrbracket \text{ in}_2
\end{align*}
\]

Combining these two we obtain an equational definition of $\text{mix}$:
\[ [p] \text{in} 1 \text{in} 2 = ([\text{mix} \; p \; \text{in} 1] \; \text{in} 2 \] \\
\text{specialized} \\
\text{program}

where if one side of the equation is defined, the other is also defined and has the same value. This is easily generalizable to various numbers of static and dynamic inputs with a more complex notation\(^2\).

Multiple language partial evaluation with different input, output, and implementation languages is also meaningful. An example is AMIX, a partial evaluator with a functional language as input, and stack code as output [20].

\[ [p]_{S} \text{in} 1 \text{in} 2 = ([\text{mix} \; L \; p \; \text{in} 1] \; \text{in} 2 \] \\
\text{specialized} \\
\text{program}

1.2 Efficiency versus Generality and Modularity?

One often has a class of similar problems which all must be solved efficiently. One solution is to write many small and efficient programs, one for each. Two disadvantages are that much programming is needed, and maintenance is difficult: a change in outside specifications can require every program to be modified.

Alternatively, one may write a single highly parameterized program able to solve any problem in the class. This has a different disadvantage: inefficiency. A highly parametrized program can spend most of its time testing and interpreting parameters, and relatively little in carrying out the computations it is intended to do.

Similar problems arise with highly modular programming. While excellent for documentation, modification and human usage, inordinately much computation time can be spent passing data back and forth and converting among various internal representations at module interfaces.

To get the best of both worlds: write only one highly parametrized and perhaps inefficient program; and use a partial evaluator to specialize it to each interesting setting of the parameters, automatically obtaining as many customized versions as desired. All are faithful to the general program, and the customized versions are often much more efficient. Similarly, partial evaluation can remove most or all the interface code from modularly written programs.

1.3 Computation in One Stage or More

Computational problems can be solved either by single stage computations, or by multistage solutions using program generation. To illuminate the problems and payoffs involved we concentrate on two familiar multistage examples:

\(^2\)Exactly the same idea applies to Prolog, except that inputs are given by partially instanted queries. In this case \( \text{in} 1 \) is the part of a query known at stage one, and \( \text{in} 2 \) instantiates this further.
1. A *compiler*, which generates a target program in some target language from a source program in a source language.

2. A *parser generator*, which generates a parser from a context free grammar.

Compilers and parser generators first transform their input into an executable program and then run the generated program, on runtime inputs for a compiler, or on a character string to be parsed. Efficiency is vital: the target program should run as quickly as possible, and the parser should use as little time per input character as possible.

Figure 3 compares two step compilative program execution with one step interpretive execution. Similar diagrams describe two step parser generation and one step general parsing.

1.3.1 Interpreters

A source program can be run in one step using an *interpreter*: an $L$-program, call it $\text{int}$, that executes $S$-programs. This has as input the $S$-program to be executed, together with its runtime inputs. Symbolically

$$\text{output} = [[\text{source}]_S \text{ in}_1 \ldots \text{ in}_n] = [[\text{int}]_L \text{ source in}_1 \ldots \text{ in}_n]$$

Formally (assuming only one input for notational simplicity), program $\text{int}$ is an *interpreter for $S$ written in $L$* if for all $\text{source}, d \in D$

$$[[\text{source}]_S d = [[\text{int}]_L \text{ source } d]$$
1.3.2 Compilers

A compiler generates a target program in target language $T$ from a source program $source$ in language $S$. The compiler is itself a program, say $compiler$, written in implementation language $L$. The effect of running $source$ on input $in_1, in_2, \ldots, in_n$ is realized by first compiling $source$ into target form:

$$\text{target} = [\text{compiler}]_L source$$

and then running the result:

$$\text{output} = [source]_S in_1 \ldots in_n = [\text{target}]_T in_1 \ldots in_n$$

Formally, $compiler$ is an $S$-to-$T$-compiler written in $L$ if for all $source$, $d \in D$,

$$[source]_S d = [[\text{compiler}]_L source]_T d$$

Comparison

Interpreters are usually smaller and easier to write than compilers. One reason is that the implementer thinks only of one time: execution time, whereas a compiler must perform actions to generate code to achieve a desired effect at run time. Another is that the implementer only thinks of one language (the source language), while a compiler writer also has to think of the target language.

Further, an interpreter, if written in a sufficiently abstract, concise and high-level language, can serve as a language definition: an operational semantics for the interpreted language.

However compilers are here to stay. The overwhelming reason is efficiency: compiled target programs usually run an order of magnitude (and sometimes two) faster than interpreting a source program.

Another source of efficiency. A two phase program may in its first phase establish global properties of its first input, and exploit them to construct a good second stage program. Examples: a compiler can type check its source program and, if type correct, generate a target program without run time checks. A parser generator may check that its input grammar is LALR(1), so allowing efficient stack-based parsing.

1.3.3 Parsing

Parsing can also be done by first generating a parser from an input context-free grammar:

$$\text{parser} = [\text{parse-gen}]_L \text{grammar}$$

and then applying the result to an input character string:

$$\text{parse-tree} = [\text{parser}]_L \text{char-string}$$

On the other hand there exist one step general parsers, e.g. Earley’s parser. Similar tradeoffs arise — a general parser is usually smaller and easier to write than a parser generator, but a parser generated from a fixed context-free grammar runs much faster.
1.4 Semantics-directed Compiler Generation

By this we mean more than just a tool to help humans write compilers. Given a specification of a programming language, for example, a formal semantics or an interpreter, our goal is automatically and correctly to transform it into a compiler from the specified “source” language into another “target” language [32,35].

Traditional compiler writing tools such as parser generators and attribute grammar evaluators are not semantics-directed, even though they can and do produce compilers as output. These systems are extremely useful in practice — but it is entirely up to their users to ensure generation of correct target code.

The motivation for automatic compiler generation is evident: thousands of man-years have been spent constructing compilers by hand; and many of these are not correct with respect to the intended semantics of the language they compile. Automatic transformation of a semantic specification into a compiler faithful to that semantics eliminates such consistency errors.

The three jobs of writing the language specification, writing the compiler, and showing the compiler to be correct (or debugging it) are reduced to one: writing the specification in a form suitable for the compiler generator.

There has been rapid progress towards this research goal in the past few years, with more and more sophisticated practical systems and mathematical theories for the semantics-based manipulation of programs. One of the most promising is partial evaluation.

1.5 Executable Specifications

A still broader goal is efficient implementation of executable specifications. Examples include compiler generation and parser generation, and others will be mentioned later.

One can naturally think of programs int and parser above as specification executers: the interpreter executes a source program on its inputs, and the parser applies a grammar to a character string. In each case the value of the first input determines how the remaining inputs are to be interpreted. Symbolically we can write:

$$\text{spec-exec}_L \text{ spec in}_1...\text{in}_n = \text{ output}$$

The interpreter’s source program input determines what is to be computed. The interpreter thus executes a specification, namely a source S-program that is to be run in language L. The first input to a general parser is a grammar that defines the structure of a certain set of character strings. The specification input is thus a grammar defining a parsing task.

A reservation is that one can of course also commit errors (sometimes the most serious ones!) when writing specifications. Achieving our goal does not eliminate all errors, but it again reduces the places they can occur to one, namely the specification. For example, a semantics-directed compiler generator allows quick tests of a new language design to see whether it is in accordance with the designers’ intentions regarding program behavior, computational effects, freedom from runtime type errors, stack usage, efficiency etc.
1.6 Generating Program Generators

In practice one rarely uses specification executers to run S-programs or to parse strings — since experience shows them often to be much slower than the specialized programs generated by a compiler or parser generator. Wouldn’t it be nice to have the best of both worlds — the simplicity and directness of executable specifications, and the efficiency of programs produced by program generators? This dream is illustrated in Figure 4:

- Program gengen accepts a two-input program p as input and generates a program generator (p-gen in the diagram).
- The task of p-gen is to generate a specialized program p_{in1}, given known value in1 for p’s first input.
- Program p_{in1} computes the same output when given p’s remaining input in2 that p would compute if given both in1 and in2.

Andrei Ershov gave the appealing name “generating extension” to p-gen. We will see that partial evaluation can realize this dream, both in theory and in practice on the computer.

Parser and Compiler Generation

Assuming gengen exists, compiler generation can be done by letting p be the interpreter int, and letting in1 be source. The result of specializing int to source is a program written in the specializer’s output language, but with the same input-output function as the source program. In other words the source program has been compiled from S into gengen’s output language. The effect is that int-gen is a compiler.
If we let \( p \) be program parser, with a given grammar as its known input \( \text{in1} \), by the description above parser-gen is a parser generator, meaning that parser-gen transforms its input grammar into a specialized parser. This application has been realized in practice at Copenhagen (unpublished as yet), and yields essentially the well-known LR(k) parsers, in program form.

**Efficiency in Program Generator Generation**

Efficiency is desirable at three different times:

1. The specialized program \( p_{\text{in1}} \) should be fast. Analogy: a fast target program
2. The program specializer \( p\text{-gen} \) should quickly construct \( p_{\text{in1}} \). Analogy: a fast compiler
3. \( \text{gengen} \) should quickly construct \( p\text{-gen} \) from \( p \). Analogy: fast compiler generation

Our goal is thus to construct an efficient program generator from a general program by completely automatic methods. On the whole the general program will be simpler but less efficient than the specialized versions the program generator produces. A telling catch phrase is *binding time engineering* — making computation faster by changing the times at which subcomputations are done.

**2 Partial Evaluation**

It would be wonderful to have a program generator generator, but it is far from clear how to construct one. Polya’s problem advice on solving hard problems: solve a simpler problem similar to the ultimate goal, and then generalize. Following this approach, we can clump boxes \( \text{gengen} \) and \( p\text{-gen} \) in Figure 4 together into a single program with two inputs, the program \( p \) to be specialized, and its first argument \( \text{in1} \). This is just the mix of Figure 1, so we already have a weaker version of the multiphase \( \text{gengen} \).

We will see how \( \text{gengen} \) can be constructed from \( \text{mix} \). This has been done in practice for several different programming languages, and efficiency criteria 1, 2 and 3 have all been met. Surprisingly, criteria 2 and 3 are achieved by *self-application* — applying the partial evaluator to itself as input.

**2.1 Speedups by Partial Evaluation**

The chief motivation for doing partial evaluation is speed: program \( p_{\text{in1}} \) is often faster than \( p \). To describe this more precisely, for any \( p, d_1, \ldots, d_n \in D \), let \( t_p(d_1, \ldots, d_n) \) be the time to compute \([p]_L d_1 \ldots d_n \). This could for example be the number of machine cycles to execute \( p \) on a concrete computer, or one could approximate by counting 1 for every elementary operation.
Specialization is clearly advantageous if \( \text{in2} \) changes more frequently than \( \text{in1} \). To exploit this, each time \( \text{in1} \) changes one can construct a new specialized \( p_{\text{in1}} \), faster than \( p \), and then run it on various \( \text{in2} \) until \( \text{in1} \) changes again. Partial evaluation can even be advantageous in a single run, since it often happens that

\[
t_{\text{mix}}(p, \text{in1}) + t_{p_{\text{in1}}} (\text{in2}) < t_p(\text{in1}, \text{in2})
\]

An analogy is that compilation plus target run time is often faster than interpretation in Lisp:

\[
t_{\text{compiler}}(\text{source}) + t_{\text{target}}(d) < t_{\text{int}}(\text{source}, d)
\]

\[2.2\] Some More Dramatic Examples

Applications of program generation include the following, all of which have been seen to give significant speedups on the computer. A common characteristic is that many involve general and rather “interpretive” algorithms. More details may be found in [25], or in the reports cited below.

**Pattern recognition.** Suppose program \( \text{recog} \) is given two inputs: a regular expression \( \text{regexp} \) and a symbol string \( \text{string} \). Its task is to determine whether \( \text{string} \) is generated by \( \text{regexp} \), i.e. to compute

\[\left[\text{recog}\right] \text{regexp string} \in \{\text{accept}, \text{reject}\}\]

In experiments the result of specializing \( \text{recog} \) to \( \text{regexp} \) is a program isomorphic to a deterministic finite automaton accepting \( \text{regexp} \) [13].

**Computer graphics.** “Ray tracing” repeatedly recomputes information about the ways light rays traverse a given scene from different origins and in different directions. Specializing a general ray tracer to a fixed scene to transform the scene into a specialized tracer, only good for tracing rays through that one scene, gives a much faster algorithm [31].

**Database queries.** Partial evaluation can compile a query into a special-purpose search program, whose task is only to answer the given query. The generated program may be discarded afterwards. Here the input to the program generator generator is a general query answerer, and the output is a “compiler” from queries into search programs [38].

**Neural networks.** Training a neural network typically uses much computer time, but can be improved by specializing a general simulator to a fixed network topology [21].

**Spreadsheets.** Spreadsheets are usually implemented interpretively, but the program generation approach has been used to transform spreadsheet specifications into faster specialized spreadsheet programs [4].
Scientific computing. General programs for several diverse applications including orbit calculations (the $n$-body problem) and computations for electrical circuits have been sped up by specialization to particular planetary systems and circuits [6].

3 Partial Evaluation and Compilation

Three main partial evaluation techniques are well known from program transformation [11]: symbolic computation, unfolding function calls, and program point specialization. The latter is a combination of definition and folding, amounting to memoisation.

Figure 2 applied the first two techniques; the third was unnecessary since the specialized program had no function calls. The idea of program point specialization is that a single function or label in program $p$ may appear in the specialized program $p_{\text{int}}$ in several specialized versions, each corresponding to data determined at partial evaluation time. An example will soon be given, and more details may be found in [25].

3.1 Compiling by Partial Evaluation

In general the idea is to specialize the interpreter to execute only one fixed source program, yielding a target program in the partial evaluator’s output language so $\text{target} = [\text{mix}] \text{int source}$.

Program $\text{target}$ can be expected to be faster than interpreting $\text{source}$ since many interpreter actions depend only on $\text{source}$ and so can be precomputed. Remark: this shows that $\text{mix}$ together with $\text{int}$ can be used to compile. It does not show that $\text{mix}$ is a compiler as defined earlier, since a compiler has only one input and $\text{mix}$ has two.

In general, program $\text{target}$ will be a mixture of $\text{int}$ and $\text{source}$, containing parts derived from both. A common pattern is that the target program’s control structure and computations resemble those of the source program, while its appearance resembles that of the interpreter, both in its language and the names of its specialized functions.

3.2 A Simple Interpreter

Several earlier articles exemplify compiling from interpreters for traditional programming languages [3,10,12,17,18,24,26,38]. An example with a different flavor but the same essence is seen in Figure 5 — a regular expression recognizer $\text{recog}$, written in a Lisp-like functional language with a (we hope) self-explanatory syntax. “Compiling” a regular expression is a way to obtain a lexical analyzer. (The underlines should be ignored for now.)

The recognizer has as inputs a regular expression $\text{regexp}$, for instance $(a+b)*abb$, and a subject string $s$. Its effect is to write #t (true) if $s$ is generated by the regular expression, else #f. The code shown accounts for the possibility that $s$ is empty. If not, its first symbol is checked against those $\text{regexp}$ could begin with, using function $\text{firstcharacters}$. If successful, the rest of $s$ is checked against a new regular expression obtained by $\text{next}$. Further explanations may be found in Bondorf’s thesis [9].
Syntax of regular expressions:

\[
\text{regexp ::= symbol | () | (regexp *) | (regexp + regexp) | (regexp regexp)}
\]

Recognizer text (Lisp-like syntax):

\[
\begin{align*}
&\text{(define (recog r s)} \cr
&\quad (\text{case s of}) \cr
&\quad \quad () : \text{(accept-empty? r)} \cr
&\quad \quad (\text{symbol \ .\ srest}) : \text{(recog1 r symbol srest (firstcharacters r)))}
\end{align*}
\]

\[
\begin{align*}
&\text{(define (recog1 r0 symbol srest firstchars)} \cr
&\quad (\text{case firstchars of}) \cr
&\quad \quad () \ : \ #f \cr
&\quad \quad (f \ .\ frest) : \text{(if (equal? symbol f)} \cr
&\quad &\quad \qquad \text{then (recog (next r0 f) srest)} \cr
&\quad &\quad \qquad \text{else (recog1 r0 symbol srest frest)))})
\end{align*}
\]

\[
\begin{align*}
&\text{(define (accept-empty? r0)} \cr
&\quad (\text{case r0 of}) \cr
&\quad \quad () \ : \ #t \cr
&\quad \quad (r1 \ *) \ : \ #t \cr
&\quad \quad (r1 \ +\ r2) : \text{(or (accept-empty? r1) (accept-empty? r2))} \cr
&\quad \quad (r1 \ r2) : \text{(and (accept-empty? r1) (accept-empty? r2))} \cr
&\quad \quad \text{else} \ : \ #f)) \quad \text{In this case r is a symbol}
\end{align*}
\]

\[
\begin{align*}
&\text{(define (next r0 f) ...)} \cr
&\text{(define (firstcharacters r0) ...)}
\end{align*}
\]

Figure 5: Regular expression recognizer

An Example

Figure 6 shows that the result of specializing “interpreter” recog with respect to “source program” source is (a+b)*abb. The “target” program is essentially the deterministic finite state automaton derived from (a+b)*abb by standard methods. An interesting observation is that mix knows nothing at all about finite automata — just how to specialize programs.

Experiments show that [target] s runs about 200 times faster than interpretively computing [recog] regexp s (this is much larger than for more traditional interpreters, where speedups of 10 are more common).
(define (recog-0 s)
  (case s of
    () : #f
    (s1.sr): (case s1 of 'a: (recog-1 sr) 'b: (recog-0 sr) else: #f))

(define (recog-1 s)
  (case s of
    () : #f
    (s1.sr):
      (case s1 of
        'a: (recog-1 sr)
        'b: (case sr of
          () : #f
          (s2.sr2):
            (case s2 of
              'a: (recog-1 sr2)
              'b: (case sr2 of
                () : #t
                (s3.sr3):
                  (case s3 of
                    'a : (recog-1 sr3)
                    'b : (recog-0 sr3)
                    else: #f))
                else: #f))
          else: #f))
    else: #f)))

Figure 6: Specialization of the Recognizer to \((a+b)\abb\)

How the target program was obtained.

The input to mix is the program recog, annotated by underlining as in Figure 5, and the
regular expression regexp. Variables r0, r1, r2, firstchars, frest and f depend only
on known input regexp, even though at run time the value of s will determine just which
values they assume during the computation. The point is that the set of all their possible
values is finite and so can be precomputed by mix during specialization.

This approach to partial evaluation proceeds by evaluating all the nonunderlined expressions in
the interpreter, and generating target code for the underlined ones. For example, expressions
involving regexp and firstchars are not underlined, while those involving s are. All function calls to recog1
are unfolded, and new specialized functions will be created where dynamic (underlined) tests occur.

In particular functions next and firstcharacters may be completely evaluated at compile time for every reachable argument combination. Following these rules gives the
target program of Figure 6.

For this fixed recog and any regexp, compilation of target = \([\text{mix}]\)recog regexp
will terminate. Proof depends on the fact that any regular expression has only finitely many “derivatives” (not hard but nontrivial).

In our experience, however, termination is usually easily established for interpretive language definitions, once the separation into static and dynamic arguments has been accomplished (by a so-called binding time analysis).

### 3.3 Partial Evaluation versus Traditional Compiling

Does partial evaluation eliminate the need to write compilers? Yes and no . . .

Given a language definition in the form of an operational semantics, partial evaluation eliminates the first and largest order of magnitude: the interpretation overhead. Further, the method yields target programs which are always correct with respect to the interpreter. Thus the problem of compiler correctness seems to have vanished.

Clearly the approach is clearly suitable for prototype implementation of new languages from interpretive definitions (known as metaprogramming in the Prolog community).

The generated target code is in the partial evaluator’s output language, typically the language the interpreter is written in. Thus partial evaluation will not devise a target language suitable for the source language, e.g. P-code for Pascal. It won’t invent new runtime data structures either, so human creativity seems necessary to gain the full handwritten compiler efficiency. Recent work by Hannan and Miller, however, suggests the possibility of deriving target machine architectures from the text of an interpreter [19].

Because partial evaluation is automatic and general, its generated code may not be as good as handwritten target code. In particular we have not mentioned classical optimization techniques such as common subexpression elimination, exploiting available expressions, and register allocation. Some of these depend on specific machine models or intermediate languages and so are hard to generalize; but there is no reason many well-known techniques could not be incorporated into the next generation of partial evaluators.

### 4 Automatic Program Generation

This section shows the sometimes surprising capabilities of partial evaluation for generating program generators.

#### 4.1 The First Futamura Projection

In Section 3.1 we saw an example of compiling by partial evaluation. This procedure always yields correct target programs, verified as follows:

\[
\text{out} = \llbracket \text{source} \rrbracket_S \text{input} \\
= \llbracket \text{int} \rrbracket \text{source input} \\
= \llbracket \text{mix} \rrbracket \text{int source input} \\
= \llbracket \text{target} \rrbracket \text{input}
\]
The last three equalities follow respectively by the definitions of an interpreter, \texttt{mix}, and \texttt{target}. The net effect has thus been to translate from \texttt{S} to \texttt{L}. Equation \( \texttt{target} = \llbracket \texttt{mix} \rrbracket \texttt{int source} \) is often called the \textit{first Futamura projection}, first reported in [16].

The conclusion is that \texttt{mix} can compile. The target program is always a specialized form of the interpreter, and so is in \texttt{mix}'s output language — usually the language in which the interpreter is written.

### 4.2 Compiler Generation by Self-application

We now show that \texttt{mix} can also generate a stand-alone compiler:

\[
\texttt{compiler} = \llbracket \texttt{mix} \rrbracket \texttt{mix int}
\]

This is an \texttt{L}-program which, when applied to \texttt{source}, yields \texttt{target}, and so a compiler from \texttt{S} to \texttt{L}, written in \texttt{L}. Verification is straightforward from the \texttt{mix} equation:

\[
\begin{align*}
\texttt{target} &= \llbracket \texttt{mix} \rrbracket \texttt{int source} \\
&= \llbracket \llbracket \texttt{mix} \rrbracket \texttt{mix int} \rrbracket \texttt{source} \\
&= \llbracket \texttt{compiler} \rrbracket \texttt{source}
\end{align*}
\]

Equation \( \texttt{compiler} = \llbracket \texttt{mix} \rrbracket \texttt{mix int} \) is called the second Futamura projection. The compiler generates specialized versions of interpreter \texttt{int}, and so is in effect \texttt{int-gen} as discussed in Section 1.6.

Operationally, constructing a compiler this way is hard to understand because it involves self-application — using \texttt{mix} to specialize itself. But it gives good results in practice, as we soon shall see.

\textbf{Remark.} This way of doing compiler generation requires that \texttt{mix} be written in its own input language, e.g. that \( \texttt{S} = \texttt{L} \). This restricts the possibility of multiple language partial evaluation as discussed in Section 1.1.

### 4.3 The Third Futamura Projection

By precisely parallel reasoning, \( \texttt{gengen} = \llbracket \texttt{mix} \rrbracket \texttt{mix mix} \) is a \textit{compiler generator}: a program that transforms interpreters into compilers. The compilers it produces are versions of \texttt{mix} itself, specialized to various interpreters. This projection is even harder to understand intuitively than the second, but also gives good results in practice. Verification of Figure 4 is again straightforward from the \texttt{mix} equation:

\[
\begin{align*}
\llbracket \texttt{p} \rrbracket \texttt{in1 in2} &= \\
\llbracket \llbracket \texttt{mix} \rrbracket \texttt{p in1} \rrbracket \texttt{in2} &= \\
\llbracket \llbracket \llbracket \texttt{mix} \rrbracket \texttt{mix p} \rrbracket \texttt{in1} \rrbracket \texttt{in2} &= \\
\llbracket \llbracket \llbracket \texttt{mix} \rrbracket \texttt{mix mix} \rrbracket \texttt{p} \rrbracket \texttt{in1} \rrbracket \texttt{in2} &= \\
\llbracket \llbracket \texttt{gengen} \rrbracket \texttt{p} \rrbracket \texttt{in1} \rrbracket \texttt{in2}
\end{align*}
\]
4.4 Speedups from Self-application

A variety of partial evaluators satisfying all the above equations have been constructed. Compilation, compiler generation and compiler generator generation can each be done in two different ways:

\[
\begin{align*}
\text{target} & = \text{[mix]} \text{ int source} \\
& = \text{[compiler]} \text{ source} \\
\text{compiler} & = \text{[mix]} \text{ mix int} \\
& = \text{[gengen]} \text{ int} \\
\text{gengen} & = \text{[mix]} \text{ mix mix} \\
& = \text{[gengen]} \text{ mix}
\end{align*}
\]

The exact timings vary according to the design of \text{mix} and \text{int}, and with the implementation language \text{L}. Nonetheless, we have often observed that in each case the second way is about 10 times faster than the first. Moral: self-application can generate programs that run faster!

4.5 Efficiency Issues

4.5.1 The Cost of Interpretation

A typical interpreter’s basic cycle is first syntax analysis; then evaluation of subexpressions by recursive calls; and finally, actions to perform the main operator, e.g. to subtract 1 or to look up a variable value. In general running time of interpreter \text{int} on inputs \text{p} and \text{d} satisfies

\[ a_p \cdot t_p(d) \leq t_{int}(p, d) \]

for all \text{d}, where \(a_p\) is a constant. (In this context, “constant” means: \(a_p\) is independent of \text{d}, but may depend on source program \text{p}.) In experiments \(a_p\) is often around 10 for simple interpreters run on small source programs, and larger for more sophisticated interpreters. Clever use of data structures such as hash tables or binary trees can make \(a_p\) grow slowly as a function of \text{p}’s size.

4.5.2 Optimality of \text{mix}

The “best possible” \text{mix} should remove all computational overhead caused by interpretation. This can be simply checked for a self-interpreter \text{sint} — an interpreter for \text{L} which is written in \text{L} (as was McCarthy’s first Lisp definition).

As above the running time of \text{sint} will be around \(a_p \cdot t_p(d)\); and \(a_p\) will be large enough to be worth reducing. Ideally \text{mix} should reduce \(a_p\) to 1. For any program \text{p} and input \text{d}

\[ [p] \text{ d} = [\text{sint}] \text{ p d} = [[[\text{mix}] \text{ sint p}] \text{ d}] \]
so \( p' = \[\text{mix}\] \text{sint} \ p \) is a program equivalent to \( p \). If \( p' \) is at least as efficient as \( p \), then all overhead caused by \( \text{sint} \)'s interpretation has been removed.

**Definition** \( \text{mix} \) is *optimal* provided \( t_{p'}(d) \leq t_p(d) \) for all \( p, d \in D \), where \( \text{sint} \) is a self-interpreter and \( p' = \[\text{mix}\] \text{sint} \ p \).

This criterion *has been satisfied* for several partial evaluators for various languages, using natural self-interpreters. In each case \( p' \) is identical to \( p \) up to variable renaming and reordering.

The same property explains the speedups resulting from self-application mentioned in the previous discussion.

### 4.6 Changing Program Style

Partial evaluation provides a novel way to construct program style transformers. Let program \( \text{int} \) be a self-interpreter for \( \text{L} \). A generated \( \text{compiler} = \[\text{gengen}\] \text{int} \) is a translator from \( \text{L} \) to \( \text{L} \) written in \( \text{L} \). In other words, it is an \( \text{L} \)-program transformer.

The transformer's output is always a specialized version of the self-interpreter. Because the basic operations used in most partial evaluators are quite simple, the output program will “inherit” many of this interpreter’s characteristics. The following examples have all been implemented this way:

1. Compiling \( \text{L} \) into a proper subset.
2. Translating direct style programs into *continuation passing style*. This is easy: just write the self-interpreter itself in continuation passing style [10].
3. Translating *lazy programs* into equivalent eager programs [26].
4. Automatic *instrumentation*, e.g. transforming a program into versions including code for step counting, or printing traces, or other debug code.

The latter idea was exploited by Shapiro to aid in debugging programs in Flat Concurrent Prolog [41].

### 4.7 Hierarchies of Metalanguages

A modern approach to solving a wide-spectrum problem is to devise a *user-oriented language* to express computational requests, viz. the widespread interest in expert systems. A processor for such a language usually works interpretively, alternating between reading and deciphering the user’s requests, consulting databases and doing problem-related computing — an obvious opportunity to optimize by partial evaluation.

Such systems are often constructed using a *hierarchy* of metalanguages, each controlling the sequence and choice of operations at the next lower level [38]. Here efficiency problems are yet more serious since each interpretation layer can multiply computation time by a significant factor.
Assume \( L_2 \) is executed by an interpreter written in language \( L_1 \), and that \( L_1 \) is itself executed by an interpreter written in implementation language \( L_0 \). The left side of Figure 7 depicts the time blowup occurring when running programs in language \( L_2 \).

**Metaprogramming without Order of Magnitude Loss of Efficiency**

The right side of Figure 7 illustrates graphically that partial evaluation can substantially reduce the cost of multiple interpretation levels. The possibility of alleviating these problems by partial evaluation has been described several places. A literal interpretation of Figure 7 involves writing two partial evaluators, one for \( L_1 \) and one for \( L_0 \). Fortunately there is an alternative approach using only a partial evaluator for \( L_0 \).

Let \( p_2 \) be an \( L_2 \)-program, and let \( \text{in}, \text{out} \) be representative input and output data, so

\[
\text{out} = ([\text{int}^1_0] L_0 \text{ int}^2_1 \ p_2 \ \text{in})
\]

One may construct an interpreter for \( L_2 \) written in \( L_0 \) as follows

\[
\text{int}^2_0 := ([\text{mix}] L_0 \text{ int}^1_0 \ \text{int}^2_1)
\]

Partial evaluation of \( \text{int}^2_0 \) can compile \( L_2 \)-programs into \( L_0 \)-programs. Better, one may construct an compiler from \( L_2 \) into \( L_0 \) by

\[
\text{comp}^2_0 := ([\text{gengen}] \text{int}^2_0)
\]

The net effect is that meta-programming may be used without order-of-magnitude loss of efficiency. Although conceptually complex, the development above has been realized in practice more than once by partial evaluation (one example is [26]), with significant speedups.
5 History and Critical Assessment

The forthcoming book [25] describes several approaches to and systems for partial evaluation. For an extensive bibliography including references to papers in Russian, see [40]. It is still being updated and is electronically available by anonymous ftp:

```
sestoft@diku.dk
```

Partial evaluation has advanced rapidly since the first years. Early systems sometimes gave impressive results, but were only applicable to limited languages, required great expertise on the part of the practitioner, and sometimes gave wrong results. Often in order to get good specialization it was necessary both to give extensive user advice on the subject program, and it was often necessary to “tune” the partial evaluator itself to fit new programs.

**Binding Time Separation**  The essence of partial evaluation is to recognize which of a program’s computations can be done at specialization time, and which should be postponed to run time. This “binding time separation” is becoming much better understood, resulting in more reliable and more powerful systems. Variants include **online** methods [6] [39] [43] in which the decision is taken “on the fly” during specialization, after the static (known) data is available; and **offline** methods [10] [12] [17] [18] which find out which computations will be performable during specialization, before the static data is available.

5.1 History

**Theory:**  The idea of obtaining a one-argument function by “freezing” an input to a two-argument function is classical mathematics (“restriction”, “projection”, or “currying”). Specializing programs rather than functions is also far from new, for instance Kleene’s s-m-n Theorem from 1936(!) is an important building block of recursive function theory. On the other hand, efficiency matters were quite irrelevant to Kleene’s investigations.

Futamura saw around 1970 that that compiling may in principle be done by partial evaluation, and that compilers may be generated by self-application [16]. Turchin, Ershov and Beckmann et.al. realized the same independently in the mid-1970s [5] [14] [42] and saw that even a compiler generator could be built by applying a partial evaluator to itself.

**Practice:**  Lombardi’s papers on incremental computation [30] were pathbreaking. In the mid-1970’s a large partial evaluator was developed in Sweden for Lisp as used in practice, including imperative features and property lists [5], and a partial evaluator for Prolog [28]. Trends to recognize partial evaluation as an important tool appeared among dedicated builders of compiler generators [32] [35].

A wide range of languages has been covered in recent years, including first order functional languages [6] [12] [24] [37], higher order languages including Scheme [10] [18] [43],
typed languages [29], logic programming including Prolog [28] [38] [39], a term rewriting language [8], and imperative languages [17] [3], including a subset of C.

Self-application: A non-trivial self-applicable partial evaluator which required hand-made unfolding annotations for function calls was first developed in late 1984 and communicated in 1985 [23]. Fully automatic self-applicable systems among the above are [3] [9] [12] [17] [18] [24] [37].

New Applications: An early motivation was optimization, so it is gratifying to see recent applications to scientific computing such as [6]. Applications to compiling and compiler generation were envisioned long before they were realized in practice, but unforeseen applications have arisen too, for example in real-time processing [33], incremental computation, debugging concurrent programs [41], and parallel and pipelined computation [36].

A potential use of fast specialization is to have a specializer running concurrently with the original program, and from time to time to switch to specialized versions whenever input patterns recur.

5.2 Critical Assessment

Partial evaluators are still far from perfectly understood in either theory or practice. Significant problems remain, and we conclude this section with some of them.

Greater Automation and User Convenience

The user should not need to give advice on unfolding or on generalization, that is to say where statically computable values should be regarded as dynamic. (Such advice is required in some current systems to avoid constructing large or infinite output programs.)

The user should not be forced to understand the logic of a program resulting from specialization. An analogy is that one almost never looks at a compiler-generated target program, or a Yacc-generated parser.

Further, users shouldn’t need to understand how the partial evaluator works. If partial evaluation is to be used by others than specialists in the field, it is essential that the user think as much as possible about the problem he or she is trying to solve, and as little as possible about the tool being used to aid its solution. A consequence is that systems and debugging facilities that give feedback about the subject program’s binding time separation are essential for use by nonspecialists.

Quite significant advances have been made, but the presence or absence of such important characteristics is all too rarely mentioned in the literature.

Analogy with Parser Generation

In several respects, using a partial evaluator is rather like using a parser generator such as Yacc. First, if Yacc accepts a grammar, then one can be certain that the parser it
generates assigns the right parse tree to any syntactically correct input string, and detects any incorrect string. Analogously, a correct partial evaluator always yields specialized programs correct with respect to the input program. For instance, a generated compiler will always be faithful to the interpreter from which it was derived.

Second, when a user constructs a context-free grammar, he or she is mainly interested in what strings it generates. But use of Yacc forces the user to think from a new perspective: possible left-to-right ambiguity. If Yacc rejects a grammar, the user may have to modify it several times, until it is free of left-to-right ambiguity.

Analogously, a partial evaluator user may have to think about his or her program from a new perspective: what are its binding time properties? If specialized programs are too slow, it will be necessary to modify the program and retry until a better binding time stage separation is achieved.

**Partial Evaluation is No Panacea**

Not all programs benefit from specialization, e.g. knowing value of $x$ will not significantly aid computing $x^n$ as in Figure 2. Further, the efficiency of mix-generated target programs depend crucially on how the interpreter is written. For example, if the interpreter uses dynamic name binding, then generated target programs will have runtime variable name searches; and if it uses dynamic code creation then generated target programs will contain runtime source language text.

### 6 Conclusions

Partial evaluation and self-application have many promising applications, and work well in practice for generating program generators, e.g. compilers and compiler generators, and other program transformers, for example style changers and instrumenters.

**Some Recurring Problems in Partial Evaluation**

Rapid progress has occurred, but there are often problems with termination of the partial evaluator, and sometimes with semantic faithfulness of the specialized program to the input program (termination, backtracking, correct answers, etc.). Further, it can be hard to predict how much (if any) speedup will be achieved by specialization, and hard to see how to modify the program to improve the speedup.

An increasing understanding is evolving of how to construct partial evaluators for various languages, of how to tame termination problems, and of the mathematical foundations of partial evaluation. On the other hand, we need to be able to

- make it easier to use a partial evaluator
- understand how much speedup is possible
- predict the speedup and space usage from the program before specialization
• deal with typed languages
• generate machine architectures tailor-made to the source language defined by an interpreter

Acknowledgements

This paper has benefited greatly from many people’s reading and constructive criticism. Special thanks are due to Carsten Gomard, Peter Sestoft and others in the TOPPS group at Copenhagen, Robert Glück, John Launchbury, Patrick O’Keefe, Carolyn Talcott, Dan Weise, and the referees.

References


## Contents

1 Introduction 1
   1.1 Partial Evaluation = Program Specialization 1
   1.2 Efficiency versus Generality and Modularity? 4
   1.3 Computation in One Stage or More 4
      1.3.1 Interpreters 5
      1.3.2 Compilers 6
      1.3.3 Parsing 6
   1.4 Semantics-directed Compiler Generation 7
   1.5 Executable Specifications 7
   1.6 Generating Program Generators 8

2 Partial Evaluation 9
   2.1 Speedups by Partial Evaluation 9
   2.2 Some More Dramatic Examples 10

3 Partial Evaluation and Compilation 11
   3.1 Compiling by Partial Evaluation 11
   3.2 A Simple Interpreter 11
   3.3 Partial Evaluation versus Traditional Compiling 14

4 Automatic Program Generation 14
   4.1 The First Futamura Projection 14
   4.2 Compiler Generation by Self-application 15
   4.3 The Third Futamura Projection 15
   4.4 Speedups from Self-application 16
   4.5 Efficiency Issues 16
      4.5.1 The Cost of Interpretation 16
      4.5.2 Optimality of mix 16
   4.6 Changing Program Style 17
   4.7 Hierarchies of Metalanguages 17

5 History and Critical Assessment 19
   5.1 History 19
   5.2 Critical Assessment 20

6 Conclusions 21