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 $\mathtt{p}_{\mathtt{in1}}$ 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]

*This work has been partly supported by the Danish Natural Sciences Research Council, and ESPRIT Basic Research Action 3124, "Semantique".

[1]Notation: data values are in ovals, and programs are in boxes. The specialized program $\mathtt{p}_{\mathtt{in1}}$ 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 $\mathtt{p}_{\mathtt{in1}}$ has only one; and $\mathtt{p}_{\mathtt{in1}}$ is the output of `mix`.

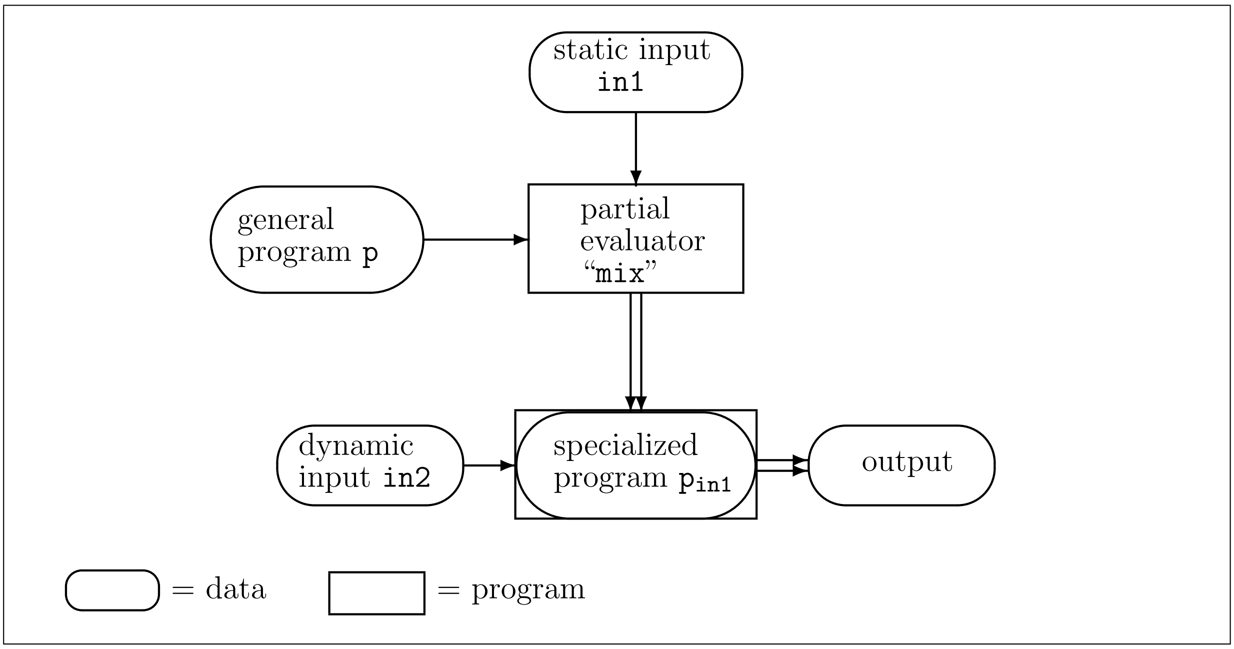

Figure 1: A Partial Evaluator

A two input program p =

```
f n x = if n = 0 then 1
        else if even(n) then f(n/2,x)↑2
        else x * f(n-1,x)
```

Program p, specialized to static input n = 5:

p_5 =

```
f5(x) = x * ((x↑2)↑2)
```

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 `in1`, and by generating code for those calculations that depend on the as yet unavailable input `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*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 `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 $D = LispAtom + D^*$, 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` =

```
(define (length x)
  (case x of
  ()            : 0
  (x1 .  xrest) : (add 1 (length xrest))))
```

We use the `typewriter` font for programs and for their input and output. If `p` is a program in language **L**, then $[\![\mathtt{p}]\!]_{\mathbf{L}}$ 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 $[\![\mathtt{p}]\!]_{\mathbf{L}} = [\![\mathtt{p}]\!]$. Standard languages used in the remainder of this article:

$\mathbf{L}$ = implementation language

$\mathbf{S}$ = source language

$\mathbf{T}$ = target language

Letting $V = D + (D \to V)$, the program meaning function $[\![\_]\!]_{\mathbf{L}}$ is of type $D \to V$. Thus

$$\mathtt{output} = [\![\mathtt{p}]\!]_{\mathbf{L}}\ \mathtt{in}_1\ \mathtt{in}_2\ \ldots\ \mathtt{in}_n$$

results from running `p` on input values $\mathtt{in}_1$, $\mathtt{in}_2$,..., $\mathtt{in}_n$, 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 `mix` is now formulated more precisely. Suppose `p` is a source program, `in1` is the data known at stage one (static), and `in2` is data known at stage two (dynamic). Then computation in one stage is described by

$$\mathtt{out} = [\![\mathtt{p}]\!]\ \mathtt{in1}\ \mathtt{in2}$$

Computation in two stages using specializer `mix` is described by

$$\begin{array}{lcl} \mathtt{p}_{\mathtt{in1}} & = & [\![\mathtt{mix}]\!]\ \mathtt{p}\ \mathtt{in1} \\ \mathtt{out} & = & [\![\mathtt{p}_{\mathtt{in1}}]\!]\ \mathtt{in2} \end{array}$$

Combining these two we obtain an equational definition of `mix`:

$$[\![\mathtt{p}]\!]\ \mathtt{in1}\ \mathtt{in2} = [\![\, \underbrace{[\![\mathtt{mix}]\!]\ \mathtt{p}\ \mathtt{in1}}_{\substack{\text{specialized}\\ \text{program}}}]\!]\ \mathtt{in2}$$

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].

$$[\![\mathtt{p}]\!]_{\mathbf{S}}\,\mathtt{in1}\,\mathtt{in2} = [\![\, \underbrace{[\![\mathtt{mix}]\!]_{\mathbf{L}}\,\mathtt{p}\,\mathtt{in1}}_{\substack{\text{specialized}\\ \text{program}}}]\!]_{\mathbf{T}}\,\mathtt{in2}$$

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 multi-stage 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 `in1` is the part of a query known at stage one, and `in2` instantiates this further.

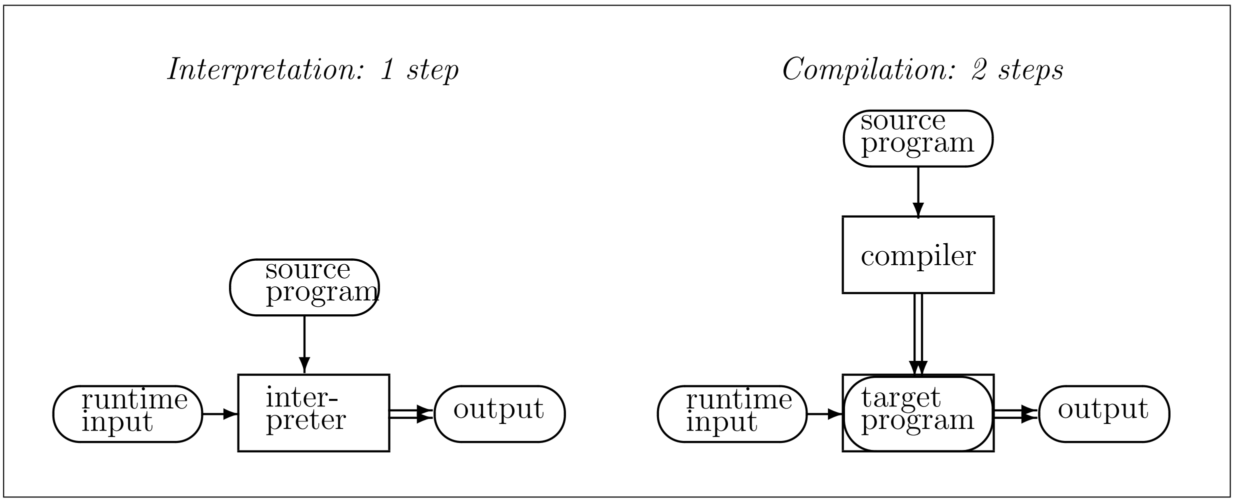

Figure 3: Compilation in Two Steps, Interpretation in One

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 `int`, that executes **S**-programs. This has as input the **S**-program to be executed, together with *its* runtime inputs. Symbolically

$$\begin{aligned} \texttt{output} &= [\![\texttt{source}]\!]_{\mathbf{S}}\ \texttt{in}_1\ \ldots\ \texttt{in}_n \\ &= [\![\texttt{int}]\!]_{\mathbf{L}}\ \texttt{source}\ \texttt{in}_1\ \ldots\ \texttt{in}_n \end{aligned}$$

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

$$[\![\texttt{source}]\!]_{\mathbf{S}}\ \texttt{d} = [\![\texttt{int}]\!]_{\mathbf{L}}\ \texttt{source}\ \texttt{d}$$

1.3.2 Compilers

A compiler generates a target program in target language $\mathbf{T}$ from a source program `source` in language $\mathbf{S}$. The compiler is itself a program, say `compiler`, written in implementation language $\mathbf{L}$. The effect of running `source` on input $\mathtt{in}_1$, $\mathtt{in}_2, \ldots$, $\mathtt{in}_n$ is realized by first compiling `source` into target form:

$$\mathtt{target} = [\![\mathtt{compiler}]\!]_{\mathbf{L}}\ \mathtt{source}$$

and then running the result:

$$\begin{aligned}\mathtt{output} &= [\![\mathtt{source}]\!]_{\mathbf{S}}\ \mathtt{in}_1\ \ldots\ \mathtt{in}_n\\ &= [\![\mathtt{target}]\!]_{\mathbf{T}}\ \mathtt{in}_1\ \ldots\ \mathtt{in}_n\end{aligned}$$

Formally, `compiler` is an $\mathbf{S}$-*to*-$\mathbf{T}$-*compiler written in* $\mathbf{L}$ if for all `source`, $\mathtt{d} \in \mathrm{D}$,

$$[\![\mathtt{source}]\!]_{\mathbf{S}}\ \mathtt{d} = [\![\ [\![\mathtt{compiler}]\!]_{\mathbf{L}}\ \mathtt{source}]\!]_{\mathbf{T}}\ \mathtt{d}$$

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 definiton: an *operational semantics* for the 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 interpretating 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:

$$\mathtt{parser} = [\![\mathtt{parse\text{-}gen}]\!]_{\mathbf{L}}\ \mathtt{grammar}$$

and then applying the result to an input character string:

$$\mathtt{parse\text{-}tree} = [\![\mathtt{parser}]\!]_{\mathbf{L}}\ \mathtt{char\text{-}string}$$

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:

$$[\![\texttt{spec-exec}]\!]_{\mathbf{L}}\ \texttt{spec}\ \texttt{in}_1 \ldots \texttt{in}_n = \texttt{output}$$

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.

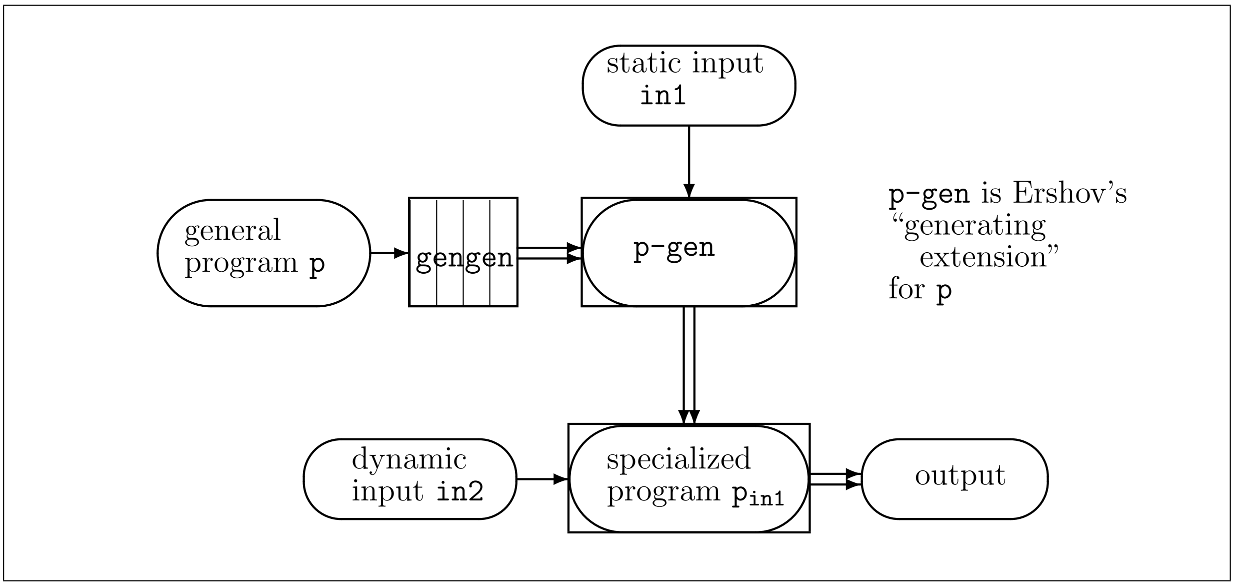

Figure 4: A Generator of Program Generators

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 $\texttt{p}_{\texttt{in1}}$, given known value `in1` for `p`'s first input.
- Program $\texttt{p}_{\texttt{in1}}$ 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 `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 $\mathtt{p_{in1}}$ should be fast. Analogy: *a fast target program*
2. The program specializer `p-gen` should quickly construct $\mathtt{p_{in1}}$. Analogy: *a fast compiler*
3. `gengen` should quickly construct `p-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 `gengen` and `p-gen` in Figure 4 together into a single program with two inputs, the program `p` to be specialized, and its first argument `in1`. This is just the `mix` of Figure 1, so we already have a weaker version of the multiphase `gengen`.

We will see how `gengen` can be constructed from `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 $\mathtt{p_{in1}}$ is often faster than `p`. To describe this more precisely, for any `p`, $\mathtt{d}_1, \ldots, \mathtt{d}_n \in D$, let $t_{\mathtt{p}}(\mathtt{d}_1, \ldots, \mathtt{d}_n)$ be the time to compute $[\![\mathtt{p}]\!]_{\mathbf{L}} \mathtt{d}_1 \ldots \mathtt{d}_n$. 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 `in2` changes more frequently than `in1`. To exploit this, each time `in1` changes one can construct a new specialized $\mathtt{p}_{\mathtt{in1}}$, faster than `p`, and then run it on various `in2` until `in1` changes again. Partial evaluation can even be advantageous in a *single run*, since it often happens that

$$t_{\mathtt{mix}}(\mathtt{p},\mathtt{in1}) + t_{\mathtt{p}_{\mathtt{in1}}}(\mathtt{in2}) < t_{\mathtt{p}}(\mathtt{in1},\mathtt{in2})$$

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

$$t_{\mathtt{compiler}}(\mathtt{source}) + t_{\mathtt{target}}(\mathtt{d}) < t_{\mathtt{int}}(\mathtt{source},\mathtt{d})$$

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 `recog` is given two inputs: a regular expression `regexp` and a symbol string `string`. Its task is to determine whether `string` is generated by `regexp`, i.e. to compute

$$[\![\mathtt{recog}]\!]\ \mathtt{regexp}\ \mathtt{string} \in \{accept, reject\}$$

In experiments the result of specializing `recog` to `regexp` is a program isomorphic to a deterministic finite automaton accepting `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 $\mathtt{p}_{\mathtt{in1}}$ 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 `target` = $[\![$`mix`$]\!]$ `int` `source`.

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

In general, program `target` will be a mixture of `int` and `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 `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 `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 possiblity that `s` is empty. If not, its first symbol is checked against those `regexp` could begin with, using function `firstcharacters`. If successful, the rest of `s` is checked against a new regular expression obtained by `next`. Further explanations may be found in Bondorf's thesis [9].

Syntax of regular expressions:

regexp ::= symbol | () | (regexp `*`) | (regexp `+` regexp) | (regexp regexp)

Recognizer text (Lisp-like syntax):

```
(define (recog r s)
   (case s of
   ()                 : (accept-empty?  r)
   (symbol . srest)   : (recog1 r symbol srest (firstcharacters r))))

(define (recog1 r0 symbol srest firstchars)
   (case firstchars of
   ()                 : #f
   (f . frest)        : (if (equal?  symbol f)
                          then (recog (next r0 f) srest)
                          else (recog1 r0 symbol srest frest))))

(define (accept-empty?  r0)
   (case r0 of
   ()                 : #t
   (r1 *)             : #t
   (r1 + r2)          : (or (accept-empty?  r1) (accept-empty?  r2))
   (r1 r2)            : (and (accept-empty?  r1) (accept-empty?  r2))
   else               : #f))               In this case r is a symbol

(define (next r0 f) ...)
(define (firstcharacters r0) ...)
```

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)))
```

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` = [[`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:

$$
\begin{aligned}
\texttt{out} &= [\![\,\texttt{source}]\!]_{\mathbf{S}}\ \texttt{input} \\
&= [\![\texttt{int}]\!]\ \texttt{source}\ \texttt{input} \\
&= [\![[\![\texttt{mix}]\!]\ \texttt{int}\ \texttt{source}]\!]\ \texttt{input} \\
&= [\![\texttt{target}]\!]\ \texttt{input}
\end{aligned}
$$

The last three equalities follow respectively by the definitions of an interpreter, `mix`, and `target`. The net effect has thus been to translate from **S** to **L**. Equation $\texttt{target} = [\![\texttt{mix}]\!]\ \texttt{int}\ \texttt{source}$ is often called the *first Futamura projection*, first reported in [16].

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

4.2 Compiler Generation by Self-application

We now show that `mix` can also generate a stand-alone compiler:

$$\texttt{compiler} = [\![\texttt{mix}]\!]\ \texttt{mix}\ \texttt{int}$$

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

$$\begin{array}{lcl} \texttt{target} & = & [\![\texttt{mix}]\!]\ \texttt{int}\ \texttt{source} \\ & = & [\![[\![\texttt{mix}]\!]\ \texttt{mix}\ \texttt{int}]\!]\ \texttt{source} \\ & = & [\![\texttt{compiler}]\!]\ \texttt{source} \end{array}$$

Equation $\texttt{compiler} = [\![\texttt{mix}]\!]\ \texttt{mix}\ \texttt{int}$ is called the second Futamura projection. The compiler generates specialized versions of interpreter `int`, and so is in effect `int-gen` as discussed in Section 1.6.

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

Remark. This way of doing compiler generation requires that `mix` be written in its own input language, e.g. that **S** = **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, $\texttt{gengen} = [\![\texttt{mix}]\!]\ \texttt{mix}\ \texttt{mix}$ is a *compiler generator*: a program that transforms interpreters into compilers. The compilers it produces are versions of `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 `mix` equation:

$$\begin{array}{ll} [\![\texttt{p}]\!]\ \texttt{in1}\ \texttt{in2} & = \\ [\![[\![\texttt{mix}]\!]\ \texttt{p}\ \texttt{in1}]\!]\ \texttt{in2} & = \\ [\![[\![[\![\texttt{mix}]\!]\ \texttt{mix}\ \texttt{p}]\!]\ \texttt{in1}]\!]\ \texttt{in2} & = \\ [\![[\![[\![[\![\texttt{mix}]\!]\ \texttt{mix}\ \texttt{mix}]\!]\ \texttt{p}]\!]\ \texttt{in1}]\!]\ \texttt{in2} & = \\ [\![[\![[\![\texttt{gengen}]\!]\ \texttt{p}]\!]\ \texttt{in1}]\!]\ \texttt{in2} & \end{array}$$

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:

$$\begin{array}{lcl}
\texttt{target} & = & [\![\texttt{mix}]\!]\ \texttt{int source} \\
 & = & [\![\texttt{compiler}]\!]\,\texttt{source} \\
 & & \\
\texttt{compiler} & = & [\![\texttt{mix}]\!]\ \texttt{mix int} \\
 & = & [\![\texttt{gengen}]\!]\ \texttt{int} \\
 & & \\
\texttt{gengen} & = & [\![\texttt{mix}]\!]\ \texttt{mix mix} \\
 & = & [\![\texttt{gengen}]\!]\ \texttt{mix}
\end{array}$$

The exact timings vary according to the design of `mix` and `int`, and with the implementation language **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 `int` on inputs `p` and `d` satisfies

$$a_{\texttt{p}} \cdot t_{\texttt{p}}(\texttt{d}) \le t_{\texttt{int}}(\texttt{p},\ \texttt{d})$$

for all `d`, where $a_{\texttt{p}}$ is a constant. (In this context, "constant" means: $a_{\texttt{p}}$ is independent of `d`, but may depend on source program `p`.) In experiments $a_{\texttt{p}}$ 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 $a_{\texttt{p}}$ grow slowly as a function of `p`'s size.

4.5.2 Optimality of `mix`

The "best possible" `mix` should remove *all computational overhead* caused by interpretation. This can be simply checked for a self-interpreter `sint` — an interpreter for **L** which is written in **L** (as was McCarthy's first Lisp definition).

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

$$[\![\texttt{p}]\!]\ \texttt{d} = [\![\texttt{sint}]\!]\,\texttt{p d} = [\![[\![\texttt{mix}]\!]\,\texttt{sint p}]\!]\ \texttt{d}$$

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

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

This criterion *has been satisfied* for several partial evaluators for various languages, using natural self-interpreters. In each case $\mathtt{p}'$ 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 int be a self-interpreter for **L**. A generated $\mathtt{compiler} = [\![\mathtt{gengen}]\!]\mathtt{int}$ is a translator from **L** to **L** written in **L**. In other words, it is an **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 **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.

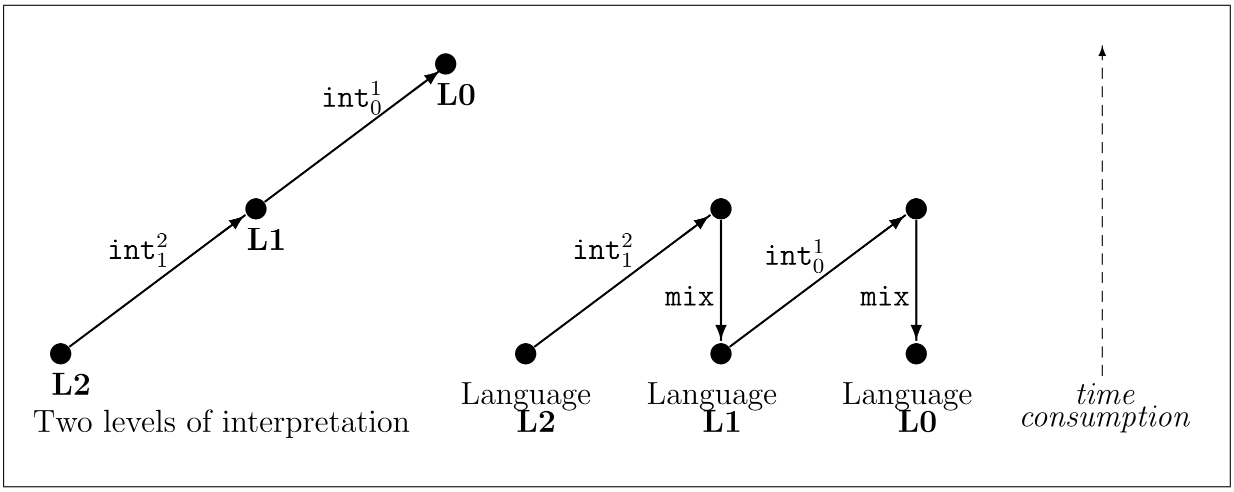

Figure 7: Overhead Introduction and Elimination

Assume **L2** is executed by an interpreter written in language **L1**, and that **L1** is itself executed by an interpreter written in implementation language **L0**. The left side of Figure 7 depicts the time blowup occurring when running programs in language **L2**.

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 **L1** and one for **L0**. Fortunately there is an alternative approach using only a partial evaluator for **L0**.

Let `p2` be an **L2**-program, and let `in`, `out` be representative input and output data, so

$$\texttt{out} = [\![\texttt{int}_0^1]\!]_{\mathbf{L0}}\ \texttt{int}_1^2\ \texttt{p2 in}$$

One may construct an interpreter for **L2** written in **L0** as follows

$$\texttt{int}_0^2 := [\![\texttt{mix}]\!]_{\mathbf{L0}}\,\texttt{int}_0^1\ \texttt{int}_1^2$$

Partial evaluation of $\texttt{int}_0^2$ can compile **L2**-programs into **L0**-programs. Better, one may construct an compiler from **L2** into **L0** by

$$\texttt{comp}_0^2 := [\![\texttt{gengen}]\!]\,\texttt{int}_0^2$$

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

[1] ACM. *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991).* ACM Press, 1991.

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

[3] L.O. Andersen. C program specialization. *International Workshop on Compiler Construction, Paderborn, Germany.* Springer-Verlag, 1992.

[4] A. Appel. Reopening Closures. Unpublished report, Princeton University, 1988.

[5] L. Beckman et al. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7(4):319–357, 1976.

[6] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[7] D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, Gammel Avernæs, Denmark, October 1987.* North-Holland, 1988. 625 pages.

[8] A. Bondorf. A self-applicable partial evaluator for term rewriting systems. In J. Diaz and F. Orejas, editors, *TAPSOFT '89. Proc. Int. Conf. Theory and Practice of Software Development, Barcelona, Spain, March 1989. (Lecture Notes in Computer Science, vol. 352)*, pages 81–95. Springer-Verlag, 1989.

[9] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

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

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

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

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

[14] A.P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[15] A.P. Ershov, D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldson, and W. Scherlis, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987 (New Generation Computing, vol. 6, nos. 2,3)*. Ohmsha Ltd. and Springer-Verlag, 1988.

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

[17] Carsten K. Gomard and Neil D. Jones, Compiler Generation by Partial Evaluation: a Case Study, *Structured Programming* 12 (1991) 123–144. Also as DIKU-report 88/24 and 90/16.

[18] C.K. Gomard and N.D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[19] J. Hannan and D. Miller. From operational semantics to abstract machines. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 323–332. ACM Press, June 1990.

[20] N.C.K. Holst. Language triplets: The AMIX approach. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 167–185. North-Holland, 1988.

[21] H.F. Jacobsen. Speeding up the back-propagation algorithm by partial evaluation. DIKU Student Project 90-10-13, 32 pages. DIKU, University of Copenhagen. (In Danish), October 1990.

[22] N.D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[23] 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. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Springer-Verlag, 1985.

[24] 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.

[25] N.D. Jones, C. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, to be published in 1993.

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

[27] U. Jørring and W.L. Scherlis. Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 86–96, 1986.

[28] 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 *Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 255–267, 1982.

[29] J. Launchbury. *Projection Factorisations in Partial Evaluation.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.

[30] 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, Cambridge, Massachusetts, 1964. MIT Press.

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

[32] P. Mosses. SIS — semantics implementation system, reference manual and user guide. DAIMI Report MD-30, DAIMI, University of Århus, Denmark, 1979.

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

[34] F.G. Pagan. *Partial Computation and the Construction of Language Processors.* Prentice-Hall, 1991. 166 pages.

[35] L. Paulson. Compiler generation from denotational semantics. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.

[36] A. Rogers K. Pingali. Compiler parallelization of simple for a distributed memory machine. In *International Conference on Parallel Programming, St. Charles IL, August 1990*, pages ?–?, 1990.

[37] S.A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.

[38] S. Safra and E. Shapiro. Meta interpreters for real. In H.-J. Kugler, editor, *Information Processing 86, Dublin, Ireland*, pages 271–278. North-Holland, 1986.

[39] D. Sahlin. The Mixtus approach to automatic partial evaluation of full Prolog. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, October 1990*, pages 377–398. MIT Press, 1990.

[40] Peter Sestoft and Alexander V. Zamulin, Annotated Bibliography on Partial Evaluation and Mixed Computation, in [15], pages 309-354, 1988. Anonymous ftp: `sestoft@diku.dk`.

[41] E. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[42] V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[43] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 165–191. ACM, Springer-Verlag, 1991.

Contents