An Introduction to Partial Evaluation

NEIL D. JONES

University of Copenhagen

Partial evaluation provides a unifying paradigm for a broad spectrum of work in program optimization, compiling, interpretation and the generation of automatic program generators [Bjørner et al. 1987; Ershov 1992; and Jones et al. 1993]. It is a program optimization technique, perhaps better called *program specialization*, closely related to but different from Jørring and Scherlis' *staging transformations* [1986]. It emphasizes, in comparison with Burstall and Darlington [1977] and Jørring and Scherlis [1986] 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.

Categories and Subject Descriptors: D.2.M [**Software Engineering**]: Miscellaneous—*rapid prototyping*; D.3.4 [**Programming Languages**]: Processors

General Terms: Experimentation, Performance, Verification

Additional Key Words and Phrases: Compilers, compiler generators, interpreters, partial evaluation, program specialization

1. INTRODUCTION

1.1 Partial Evaluation = Program Specialization

In Figure 1 a partial evaluator `mix` is given a subject program `p` 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.[1]

Correctness of $\mathtt{p}_{\mathtt{in1}}$ can be described equationally: for this `p`, `in1`, and all `in2`

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

In other words a partial evaluator is a *program specializer*.

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

[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; $\mathtt{p}_{\mathtt{in1}}$ is the output of `mix`.

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

Author's address: DIKU, Dept. of Computer Science., Univ. of Copenhagen Universitetsparken 1, DK-2100 Copenhagen East, Denmark. ⟨email:neil@diku.dk⟩.

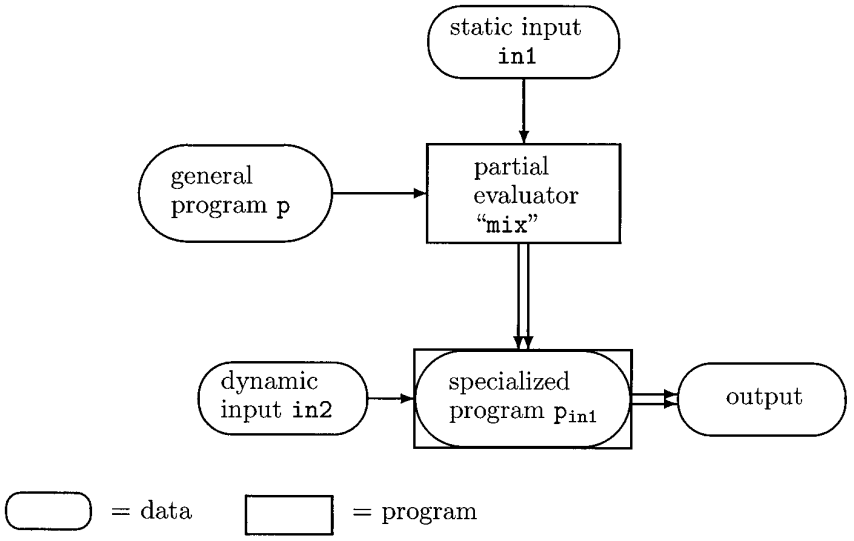

Figure 1. A partial evaluator.

on the as yet unavailable input `in2`. Figure 2 shows a program to compute x^n, and a faster program p_5 resulting from specializing `p` to $n = 5$ (ignore the underlines for now).

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, then 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" [Ershov 1982], 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 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.

We use the `typewriter` font for programs and for their input and output. If `p` is a program in language `L`, then $[\![\texttt{p}]\!]_{\texttt{L}}$ denotes its meaning—typically a function from several inputs to an output. The subscript `L` indicates how `p` is to be interpreted. When only one language is being discussed we often omit the subscript so $[\![\texttt{p}]\!]_{\texttt{L}} = [\![\texttt{p}]\!]$. Standard languages used in the remainder of this article are:

`L`: an implementation language
`S`: a source language
`T`: a target language

The program meaning function $[\![\_]\!]_{\texttt{L}}$ is of type $D^* \rightarrow V$. Thus

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

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

The Defining Equation. The essential property of a partial evaluator mix is now formulated more precisely. Suppose `p` is a source program expecting two inputs, `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

```
out=[[p]] [in1, in2]
```

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.

Computation in two stages using specializer mix (as in Figure 1) is described by

$$\texttt{pin1} = [\![\texttt{mix}]\!]\ [\texttt{p, in1}]$$
$$\texttt{out} = [\![\texttt{p}_{\texttt{in1}}]\!]\ \texttt{in2}$$

Combining these two we obtain an equational definition of mix:

$$[\![\texttt{p}]\!]\ [\texttt{in1, in2}] = [\![[\![\texttt{mix}]\!]\ [\texttt{p, in1}]]\!]\ \texttt{in2}$$

specialized program

Here equality needs a broader interpretation than usual: it means that if one side of the equation is defined, then the other side 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 (say S, L, T, respectively) is also meaningful. An example is AMIX, a partial evaluator with a functional language as input and stack code as output [Holst 1988]:

$$[\![\texttt{p}]\!]_{\texttt{S}}[\texttt{in1, in2}] = [\![[\![\texttt{amix}]\!]_{\texttt{L}}[\texttt{p, in1}]]\!]_{\texttt{T}}\ \texttt{in2}$$

specialized program

[2] Exactly the same idea applies to Prolog, except that inputs are given by partially instantiated queries and answers are sequences of terms as variable values. In this case in1 is the part of a query known at stage one and in2 instantiates this query further.

1.2 Speedups by Partial Evaluation

The chief motivation for doing partial evaluation is speed: program p_{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 machine code for p on a concrete computer.

Specialization is clearly advantageous if in2 changes more frequently than in1. To exploit this, each time in1 changes one can construct a new specialized p_{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

$$t_{mix}(p, in1) + t_{p_{in1}}(in2) < t_p(in1, in2)$$

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

$$t_{\text{comp}}(src) + t_{\text{targ}}(d) < t_{\text{interp}}(src, d)$$

2. HOW CAN PARTIAL EVALUATION BE DONE?

We use the term "partial evaluation" for *automatic* program specialization—*not* hand-directed program transformation or verification. Three main partial evaluation techniques are well known from program transformation [Burstall and Darlington 1977]: *symbolic computation*, *unfolding* function calls, and *pro-*

A two input program $p =$

```
a(m,n) = if m = 0 then n+1 else
           if n = 0 then a(m-1,1) else
         a(m-1,a(m,n-1))
```

Program p, specialized to static input m = 2:

$p_2 =$

```
a2(n) = if n=0 then a1(1) else a1(a2(n-1))
a1(n) = if n=0 then a0(1) else a0(a1(n-1))
a0(n) = n+1
```

Figure 3. Specialization of a program for Ackermann's function.

gram point specialization. The latter is a combination of *definition creation* and *folding*, amounting to *memoization*.

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_{in1} in several specialized versions, each corresponding to data determined at partial evaluation time. More details may be found in Jones et al. [1993].

A representative example. Ackermann's function is useless for practical computation but an excellent vehicle to illustrate program point specialization. An example is seen in Figure 3 (the underlines should still be ignored). Note that the specialized program uses *less than half as many* arithmetic operations as the original.

2.1 Online and Offline Specialization

Figure 3 illustrates *offline* specialization [Bondorf 1991; Consel 1993; Gomard and Jones 1991a; 1991b]. This makes use of program *annotations*, here indicated by underlines, e.g., n-1. These can be regarded as instructions to the specializer. Offline specialization begins with a so-called *binding-time analysis*, whose task is to place appropriate annotations on the program before reading the static input.

Interpretation of the annotations by the specializer is simple:

(1) *evaluate* all nonunderlined expressions;
(2) *unfold at specialization time* all nonunderlined function calls;
(3) *generate residual code* for all underlined expressions; and
(4) *generate residual function calls* for all underlined function calls.

An alternative, called *online* specialization, computes program parts as early as possible and takes decisions "on the fly" using only (and all) available information [Berlin and Weise 1990; Sahlin 1990; Weise et al. 1991].

Figure 3 is a clear improvement over the unspecialized program, but can obviously be improved even more by "on-the-fly" reductions to

```
a2(n) = if n = 0 then 3
                 else a1(a2(n − 1))

a1(n) = if n = 0 then 2
                 else a1(n − 1) + 1
```

These methods often work better than offline methods, in particular on structured data that is partially static and

partially dynamic. On the other hand, they introduce new problems and new techniques concerning termination of specializers. Comparisons and evaluations of costs and benefits can be found in Ruf [1993] and Jones [1993, Ch. 7].

2.2 Sketch of an Offline Partial Evaluator

Consider as given (1) a first-order functional program of form

```
f1(s, d) = expression1
g(u, v, · · ·) = expression2
· · ·
h(r, s, · · ·) = expressionm
```

and (2) *annotations* that mark every function parameter, operation, test, and function call as either *eliminable*: perform or compute or unfold during specialization, or *residual*: generate program code to appear in the specialized program.

In particular, the parameters of any definition of a function `f` can be partitioned into those which are *static* and the rest, which are *dynamic*. For instance, `m` is static and `n` is dynamic in the Ackermann example.

The specialized program will have the same form as the original, but it will consist of definitions of *specialized functions* (program points) $g_{Statvalues}$, each corresponding to a pair (`g, Statvalues`) where `g` is defined in the original program and `Statvalues` is a tuple consisting of some values for all the static parameters of `g`. The parameters of function $g_{Statvalues}$ in the specializer will be the remaining parameters of `g`, all dynamic.

Finally, we give the specialization algorithm sketch, assuming the input program's defining function is given by `f1(s, d) = expression1` and that `s` is static and `d` is dynamic. In the following, variables `Seenbefore` and `Pending` both range over sets of specialized functions $g_{Statvalues}$. Specializer output variable `Target` will always be a list of (residual) function definitions.

1. Read `Program` and `S`.
2. `Pending` := {$f1_s$}; `Seenbefore` :={};
3. `While Pending` ≠ {} do 4—6:
4. Choose and remove a pair $g_{Statvalues}$ from `Pending`, and add it to `Seenbefore` if not already there.
5. Find `g`'s definition `g(x1, x2, . . .) = g-expression` and let `D1`, . . . , `Dm` be all its dynamic parameters.
6. Generate and append to Target the definition

$$g_{Statvalues}(D1, \cdot\ \cdot\ \cdot, Dm) = \mathit{Reduce}(E);$$

where `E` is the result of substituting static value `Si` from `Statevalues` in place of each static g-parameter `xi` occurring in `g-expression`

Reduction of an expression `E` to its residual equivalent `RE` = *Reduce*(`E`) is defined by:

1. If `E` is constant or a dynamic parameter of `g`, then `RE = E`.
2. If `E` is a static parameter of `g` then `RE` = its value, extracted from the list `Statvalues`.
3. If `E` is of form `operator(E1, . . . , En)` then compute the values $v_1, \ldots, v_n$ of *Reduce*(`E1`), . . . , *Reduce*(`En`).
   (These must be totally computable from `g`'s static parameter values, else the annotation is in error.)
   Then set `RE` = the value of `operator` applied to $v_1, \ldots, v_n$.
4. If `E` is `operator(E1, . . . , En)` then compute `E1′` = *Reduce*(`E1`), . . . , `En′` = *Reduce*(`En`).
   Then set `RE` = the expression "`operator(E1′, . . . , En′)`".
5. If `E` is `if E0 then E1 else E2` then compute *Reduce*(`E0`). This must be constant, else the annotation is in error. If *Reduce*(`E0`) equals `true`, then `RE` = *Reduce*(`E1`), otherwise `RE` = *Reduce*(`E2`).
6. If `E` is `if E0 then E1 else E2` then `RE` = the expression "`if E0′ then E1′ else E2′`" where each `Ei′` equals *Reduce*(`Ei`).

7. Suppose `E` is `f(E1, E2, . . . , En)` and `Program` contains definition `f(x1 . . . xn) = f-expression` then `RE` = *Reduce*(`E′`), where `E′` is the result of substituting *Reduce*(`Ei`) in place of each static `f`-parameter `xi` occurring in `f-expression`.
8. If `E` is `f(E1, E2, . . . , En)`, then
   (a) Compute the tuple `Statvalues′` of the static parameters of `f`, by calling *Reduce* on each. This will be a tuple of constant values (if not, the annotation is incorrect).
   (b) Compute the tuple `Dynvalues` of the dynamic parameters of `f`, by calling *Reduce*; this will be a list of expressions.
   (c) Then `RE` = the call $f_{Statvalues'}$`(Dynvalues)`
   (d) A side-effect: if $f_{Statvalues'}$ is neither in `Seenbefore` nor in `Pending`, then add it to `Pending`.

2.3 Congruence, Binding-Time Analysis, and Finiteness

Where do the annotations used by the algorithm above come from? Their root source is knowledge of which inputs will be known when the program is specialized, for example `m` but not `n` in the Ackermann example. There are two further requirements for the specialization algorithm above to succeed.

First, the internal parts of the program must be properly annotated (witness comments such as "if . . . the annotation is incorrect"). The bottom line is that if any parameter or operation has been marked as eliminable, then one needs a guarantee that it actually will be so when specialization is carried out, for *any possible static program inputs*. For example, an if marked as eliminable must have a test part that evaluates to a constant. This requirement (properly formalized) is called the *congruence* condition.

The second condition is *termination*: regardless of what the values of the static inputs are, the specializer should attempt to produce neither infinitely many residual functions nor any infinitely large residual expression.

It is the task of *binding-time analysis* to ensure that these conditions are satisfied. Given an unmarked program together with a division of its inputs into static (will be known when specialization begins) and dynamic, the binding-time analyzer proceeds to annotate the whole program. Several techniques for this are described in Jones et al. [1993].

The current state of the art is that congruence can definitely be achieved automatically, whereas binding-time analyses that guarantee termination are only beginning to be constructed. The problem is complex in that the binding-time analysis must account for possible consequences not one step into the future, but two.

3. COMPILERS, INTERPRETERS

3.1 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 describe two familiar examples, at first informally:

(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 4 compares two-step compilative program execution with one-step interpretive execution. Similar diagrams describe two-step parser generation and one-step general parsing.

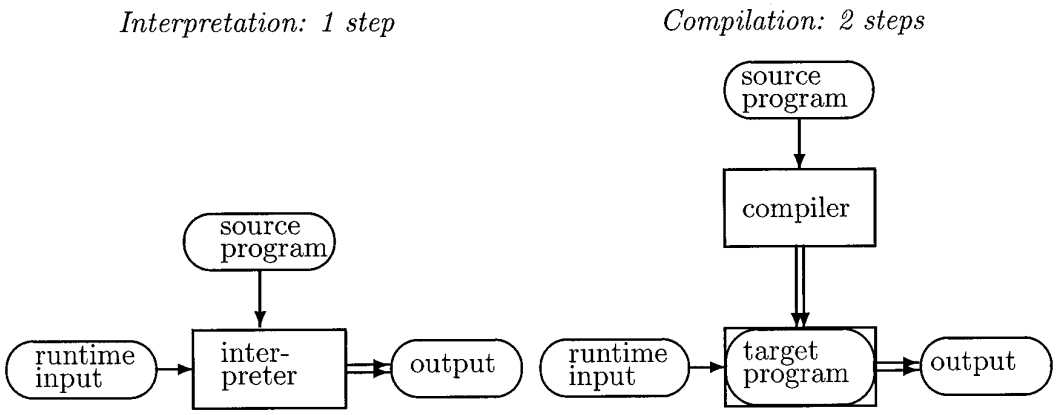

Figure 4. Compilation in two steps, interpretation in one.

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 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.

3.2 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* run-time inputs. Symbolically

$$\begin{aligned} \mathtt{ou8tput} &= [\![\mathtt{source}]\!]_{\mathtt{S}}\ [\mathtt{in}_1, \cdots, \mathtt{in}_n] \\ &= [\![\mathtt{int}]\!]_{\mathtt{L}}\ [\mathtt{source}, \mathtt{in}_1, \ldots, \mathtt{in}_n] \end{aligned}$$

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

$$[\![\mathtt{source}]\!]_{\mathtt{S}}\ \mathtt{d} = [\![\mathtt{int}]\!]_{\mathtt{L}}\ [\mathtt{source}, \mathtt{d}]$$

3.3 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 $\mathtt{in}_1, \mathtt{in}_2, \ldots, \mathtt{in}_n$ is realized by first compiling source into target form:

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

and then running the result:

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

Source program `src`:

```
((IFX=0 5) Y:=Y+1 Y:=Y+1 X:=X-1(GOTO 0))
```

Interpreter `interp`:

```
Execute (pgm, x) = Run (pgm, pgm, x, 0)

Run (pgm, pgmtail, x, y) =
let (instr . rest) = pgmtail in
case instr of
'X:=X+1 :  Run (pgm, rest, x+1, y)
'X:=X-1 :  Run (pgm, rest, x-1, y)
'Y:=Y+1 :  Run (pgm, rest, x, y+1)
'Y:=Y-1 :  Run (pgm, rest, x, y-1)
('GOTO l):  Run( pgm, Nth(l, pgm), x, y))
('IFX=0GOTO l) :
  if x = 0 then Run( pgm, (Nth(l,pgm), x, y)
    else Run (pgm, rest, x, y))
('IFY=0GOTO l) :
  if y = 0 then Run( pgm, (Nth(l,pgm), x, y)
    else Run (pgm, rest, x, y))
else ERROR)))
```

Specialized interp. `tgt`:

```
Execute-1 (x) = Run-1 (x, 0))
Run-1 (x, y) = if x=0 then y else
                      Run-1 (x-1, (y+1)+1)
```

Figure 5. Functional interpreter for an imperative language.

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

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

4. COMPILING BY SPECIALIZATION

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 that

```
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.

4.1 Example of Compiling

We now consider a fairly trivial interpreter written *for* an imperative language S *in* a simple functional language L. A source program is an instruction sequence with the following syntax (only seven instructions!):

```
X:=X+1, X:=X−1,
Y:=Y+1,  Y:=Y−1,
(IFY=0GOTO label),
(IFX=0GOTO label),
(GOTO label)
```

The middle box of Figure 5 contains an interpreter `interp` for `S` written in `L`, using a syntax whose meaning, we hope, is obvious.

How the interpreter works. the "instruction counter" is maintained by argument `pgmtail` of `Run`. Its value is always a suffix of source program `pgm`, and its first instruction is the next to be executed. Normal sequential execution just removes this first instruction and contains with the remainder, called `rest` in `Run`. Control transfer is handled by function `N-th` which, given a label ℓ, finds the ℓ-th instruction in `pgm` and returns the suffix of `pgm` beginning at that point. (This is the reason that argument `pgm` is passed along throughout the interpreter.)

Effect of specialization. Figure 5 shows in the uppermost box a source program `src` that doubles its input `x` by counting `y` up by 2 each time it counts `x` down by 1, stopping when `x` becomes zero. (Instructions are labeled 0, 1, . . . , so 5 denotes the end of the program.) The lowermost box shows a functional target program `tgt` equivalent to source program `src`. Our claim is that specialization of `interp` with respect to `src` yields `tgt`.

The underline in the call in `Execute` causes generation of the call to `Run-1`. (This is function `Run`, specialized to `src`, `src` as its first two arguments.)

Within function Run, all the "syntactic dispatch" operations (the `case, let` and patterns) having to do with source program structure can be done statically, and so do not appear at all in the target program. On the other hand, operations involving `x` and `y` cannot be so executed, so code is generated (residual target code `x − 1` and `(y + 1) + 1`).

As to function calls of Run to itself, those which decrease argument `pgmtail` (i.e., those reflecting normal sequential execution) may *safely be unfolded*, since `pgmtail` can only be decreased finitely many times (`pgmtail` is a list, not an integer). The remaining calls (to interpret tests and GOTOs) may *not* be safely unfolded—this could cause infinite unfolding and thus nonterminating specialization in the case of source program loops. They are thus marked (by underlining) as "not to be unfolded," and thus account for the call to `Run-1` nested inside `Run-1`.

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.

4.2 Partial Evaluation Versus Traditional Compiling

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 (assuming, of course, that `mix` is correct). Thus the problem of compiler correctness seems to have vanished.

Clearly the approach is suitable for prototype implementation of new languages which are defined interpretively, as has been done in functional languages since very early times [McCarthy et al. 1962].

The generated target code is in the partial evaluator's output language, typically the language in which the interpreter is written. Thus partial evaluation will not devise a target language tailor-made to 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 [1990], however, suggests the possibility of deriving target machine architectures from the text of an interpreter.

Because partial evaluation is *automatic and general*, its generated code may not be as good as handwritten tar-

get 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.

4.3 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 do arithmetic operations or a variable lookup. In general, running time of interpreter int on inputs p and d satisfies

$$a_p \cdot t_p(d) \leq t_{int}(\mathtt{p},\ \mathtt{d})$$

for all d, where a_p is a constant. (In this context, "constant" means: a_p is independent of d, but may depend on source program 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 p's size.

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 $a_p \cdot t_p(d)$; and a_p will be large enough to be worth reducing. Ideally mix should reduce a_p to 1. For any program p and input d

$$[\![p]\!]\ d = [\![sint]\!]\ [p,\ d]$$
$$= [\![[\![mix]\!]\ [sint,\ p]]\!]d$$

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

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

This criterion *has been satisfied* for several partial evaluators for various languages, using natural self-interpreters [Jones et al., p. 174; Romanenko 1988]. 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.

4.4 Example with Larger Speedup

Several earlier articles exemplify compiling from interpreters for traditional programming languages [Andersen 1992; Bondorf 1991; Consel 1993; Gomord and Jones 1991a; 1991b; Jones et al. 1989; Jørgensen 1992; Safra and Shapiro 1986]. An example with a different flavor but the same essence is seen in Figure 6—a regular expression recognizer rex, written in a Lisp-like functional language. "Compiling" a regular expression is a way to obtain a lexical analyzer.

The recognizer rex has as inputs a regular expression r, e.g., (a + b)* abb, and a subject string s. The recognizer's effect is to return #t (true) if s is generated by the regular expression, else #f. The code shown takes into account the possibility that s is empty. If not, its first symbol is checked against those r 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 [1990].

For example, suppose r = (a + b)* abb. The results of some calls follow:

```
(accept-empty? r) #f
(next r 'a)       bb + (a + b)*abb
(next r 'b)       (a + b)*abb
```

Syntax of regular expressions:

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

Recognizer text (Lisp-like syntax):

```
(define (rex r s)
   (case s of
   ()              : (accept-empty?  r)
   (symbol . srest): (rex1 r symbol srest (firstcharacters r))))

(define (rex1 r0 symbol srest firstchars)
   (case firstchars of
   ()              : #f
   (f . frest)     : (if (equal?  symbol f)
                       then (rex (next r0 f) srest)
                       else (rex1 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 6. Regular expression recognizer.

An example target program. Figure 7 shows that the result of specializing "interpreter" `rex` with respect to "source program" is `(a + b)* abb`.

The "target" program is essentially a program form of 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 in general ⟦`target`⟧ `s` runs about 200 times faster than interpretively computing ⟦`rex`⟧ `regex s`. This is much larger than for more traditional interpreters, where speedups of 10 are more common.

How the target program was obtained. The input to `mix` is the program `rex`, whose dynamic parts are annotated by underlining as in Figure 6, and the regular expression `r`. Variables `r0, r1, r2, firstchars, frest` and `f` depend only on known (static) input `r`, 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.

In particular, functions `next` and `first-characters` may be completely evaluated at compile time for every reachable argument combination. Following these rules gives the target program of Figure 7.

For this fixed `rex` and any r, compilation of `target` = ⟦`mix`⟧`[rex, r]` will

```
(define (rex-0 s)
  (case s of
    ()     : #f
    (s1.sr): (case s1 of 'a: (rex-1 sr) 'b: (rex-0 sr) else: #f)

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

Figure 7. Specialization of the recognizer to (a+b)*abb.

terminate. Proof depends on the fact that any regular expression has only finitely many "derivatives" (not hard but nontrivial to prove).

5. GENERATION OF PROGRAM GENERATORS

Why not take our own medicine and apply partial evaluation to produce faster generators of specialized programs? A telling catch phrase is *binding-time engineering*—making computation faster by changing the times at which subcomputations are done.

This can indeed be done, yielding a *generator of program generators* as in Figure 8. Efficiency of such a procedure is desirable at three different times:

(1) The specialized program p_{in1} should be fast. Analogy: *a fast target program*.

(2) The program specializer `p-gen` should quickly construct p_{in1}. Analogy: *a fast compiler*.

(3) `cogen` 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.

5.1 Generating Program Generators

In practice one rarely uses extremely general programs, e.g., specification executers, to run programs or to parse strings—since experience shows them often to be much slower than the spe-

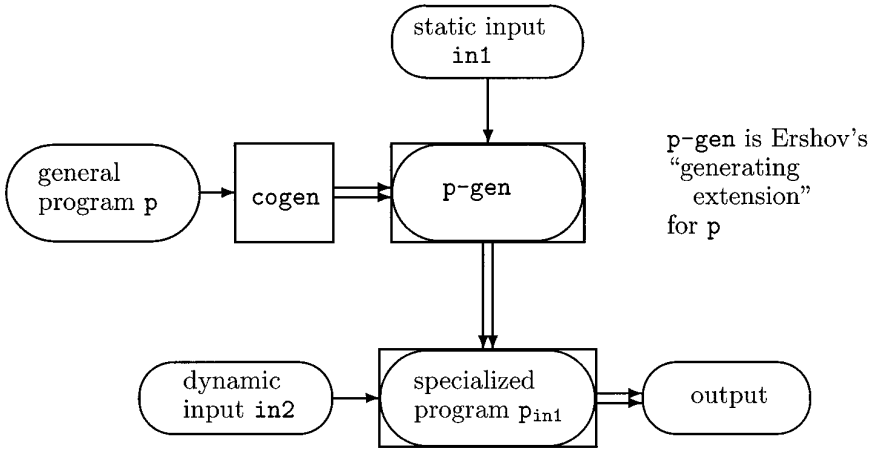

Figure 8. A generator of program generators.

cialized 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 8:

—Program `cogen` 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` [1982]. We will see that partial evaluation can realize this dream quite generally, both in theory and in practice on the computer.

First, we give an example to demystify Figure 8 by showing a generating extension for our very first example, the exponentiation program from Figure 2. This is seen in Figure 9. Since the purpose of `p-gen` is to generate program code, we have used an informal string notation with quotes for output code.

Efficiency in program generator generation. It would be wonderful to have such a tool, but it is far from clear how to construct one. Polya's problem advice on solving hard problems was to solve a simpler problem similar to the ultimate goal, and then to generalize. Following this approach, we can clump boxes `cogen` and `p-gen` in Figure 8 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 `cogen`.

We will see how `cogen` 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.

5.2 Compiling by the First Futamura Projection

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

In Section 4 we saw examples of com-

Generating extension for exponent program p-gen =

```
f-gen(n) = 'h(x) =' ++ g(n)

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

Program p, specialized to static input n = 19:

$p_{11} = [\![p\text{-}gen]\!](19) =$

```
h(x) = (x * (x * x↑2↑2↑2)↑2)
```

Figure 9. Generating extension for a program to compute x^n.

piling by partial evaluation. This procedure always yields correct target programs, verified as follows:

```
out = [[source]]_S input
    = [[int]] [source, input]
    = [[[mix]] [int, source]] input
    = [[target]] input
```

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

```
target = [[mix]][int, source]
```

is often called the *first Futamura projection*, first reported in Futamura [1971].

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.

5.3 Compiler Generation the Second Futamura Projection

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

```
compiler = [[mix]] [mix, int]
```

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

```
target = [[mix]][int, source]
       = [[[[mix]][mix, int]] source
       = [[compiler]] source
```

Equation compiler = [[mix]][mix, 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 5.1. A concrete example may be seen in Jones et al. [1993, Ch. 4].

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.

5.4 Compiler Generator Generation by the Third Futamura Projection

By precisely parallel reasoning, cogen = [[mix]][mix, 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 8 is again straightforward from the mix equation:

```
⟦p⟧ [in1, in2] =
⟦⟦mix⟧ [p, in1]⟧ in2 =
⟦⟦⟦mix⟧ [mix, p]⟧ in1⟧ in2 =
⟦⟦⟦⟦mix⟧ [mix, mix]⟧ p⟧ in1⟧ in2 =
⟦⟦⟦cogen⟧ p⟧ in1⟧ in2
```

5.5 Speedups by 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:

```
target   = ⟦mix⟧ [int, source]
         = ⟦compiler⟧ source
compiler = ⟦mix⟧ [mix, int]
         = ⟦cogen⟧ int
cogen    = ⟦mix⟧ [mix, mix]
         = ⟦cogen⟧ mix
```

The exact timings vary according to the design of mix and int, and with the implementation language L. Nonetheless, a number of researchers have observed that *in each case the second way is often about 10 times faster than the first* [Consel 1993; Gomard and Jones 1991b; Jones et al. 1993; Jones et al. 1989; Romanenko 1988]. Moral: self-application can generate programs that run faster!

Parser and Compiler Generation. Assuming cogen 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 cogen'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 [Sperber and Thiemann, 1995] and yields essentially the well-known LR(k) parsers, in program form.

6. AUTOMATIC PROGRAM GENERATION

6.1 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 compiler = ⟦cogen⟧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) Automatic *instrumentation*, e.g., transforming a program into versions including code for step counting, or printing traces, or other debug code.
(3) Translating direct style programs into *continuation-passing style*. This is easy: just write the self-interpreter itself in continuation-passing style [Bondorf 1991].
(4) Translating *lazy programs* into equivalent eager programs [Jørgensen 1992].
(5) Translating direct-style programs into *tail-recursive style* suitable for machine code implementation. In principle this can be done by "just" writing the self-interpreter itself in tail-recursive style, but achieving

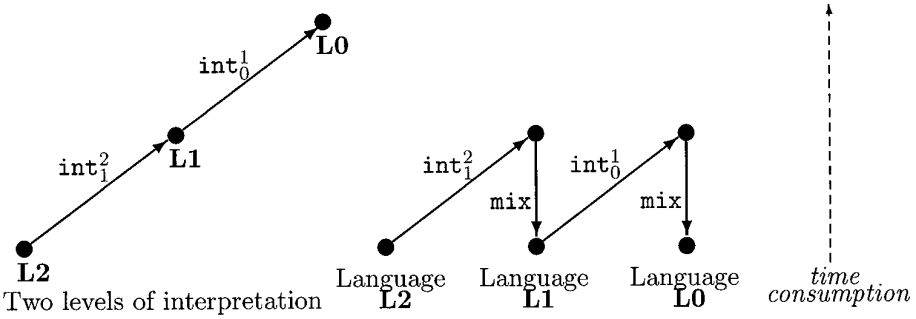

Figure 10. Overhead introduction and elimination.

the desired binding-time separation is a quite nontrivial task [Sperber and Thiemann 1996].

Example (2) was exploited by Shapiro [1983] to aid in debugging programs in Flat Concurrent Prolog.

6.2 Hierarchies of Metalanguages

A modern approach to solving a wide-spectrum problem is to devise a *user-oriented language* to express computational requests, hence, 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 [Safra and Shapiro 1986]. Here efficiency problems are yet more serious since each interpretation layer can multiply computation time by a significant factor.

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 10 depicts the time blowup occurring when running programs in language L2.

Metaprogramming without order-of-magnitude loss of efficiency. The right side of Figure 10 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 in several places. A literal interpretation of Figure 10 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

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

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

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

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

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

The net effect is that metaprogramming 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 Jørgensen [1992]), with significant speedups.

6.3 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 [Mosses 1979; Paulson 1984].

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.

6.4 Executable Specifications

A still broader goal is *efficient implementation of executable specifications*. Examples include compiler generation and parser generation.

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:

$$[\![\texttt{spec-exec}]\!]_{\texttt{L}} \quad [\texttt{spec}, \texttt{in}_1, \cdots, \texttt{in}_n] = \texttt{output}$$

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 at which 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 run-time type errors, stack usage, efficiency, etc.

7. BROADER PERSPECTIVES

Partial evaluation is no panacea. Program specialization, like highly optimizing compilation, may not be worthwhile for all applications. For one example, knowing the 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 another, if an interpreter uses *dynamic variable name binding*, then generated target programs will have run-time variable name searches; and if it uses *dynamic source code creation* then generated target programs will contain run-time source language text.

Characteristics of a problem that make it suitable for specialization include: It is *time-consuming*. It must be solved *repeatedly*. It is often solved with *similar parameters*, for example: code

keys in cryptography, a circuit to simulate, a query to evaluate, or a network communication pattern. It is solved by *well-structured* and *cleanly written* software (programming tricks make it hard, not only for the human to maintain, but also for the specializer). It implements a *high-level applications-oriented* language.

7.1 Efficiency Versus Generality and Modularity?

One often has a class of similar problems that 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 parametrized program able to solve any problem in the class. This has a different disadvantage: *inefficiency*. A highly parameterized 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 parameterized 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.

7.2 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 Jones et al. [1993] or in the reports cited below.

Pattern recognition. An earlier example gave the result of specializing a general regular expression recognizer to a particular expression, with dramatic speedup. This theme has been carried much further, for several classes of patterns, by several researchers [Consel and Danvy 1989; Danvy 1991].

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 [Andersen 1994; Mogensen 1986].

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 is a general query answerer and the output is a "compiler" from queries into search programs [Safra and Shapiro 1986]. A more recent application is specialized integrity checks through partial evaluation of meta-interpreters [Leusche and De Shreye 1995].

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 [Jacobsen 1990].

Spreadsheets. Spreadsheets are usually implemented interpretively, but the program generation approach has been used to transform spreadsheet specifica-

tions into faster specialized spreadsheet programs [Appel 1988].

Scientific computing. General programs for several diverse applications including orbit calculations (the *n*-body problem) and computations for electrical circuits have been speeded up by specialization to particular planetary systems and circuits [Baier et al. 1994; Berlin and Weise 1990].

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

$$\texttt{parser} = [\![\texttt{parse-gen}]\!]_{\texttt{L}}\ \texttt{grammar}$$

and then applying the result to an input character string:

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

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.

8. AUTOMATION AND PARTIAL EVALUATION

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.

In our experience it is usually fairly easy to establish termination, when given a particular interpretive language definition and when the separation into static and dynamic arguments have been accomplished by a binding-time analysis. Ensuring termination may, however, require small changes to the interpreter. Binding-time analysis sufficient to give a congruent separation is now well automated; but fully automatic binding-time analyses sufficiently conservative to guarantee termination of specialization, or to perform "binding-time improvement" program transformations automatically, are still topics of ongoing research.

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.

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 to "tune" the partial evaluator itself to fit new programs.

Need for human understanding. A deeper difficulty is that new problems, and programs that solve them, require *understanding*, a program development aspect never likely to be fully automated (regardless of progress in artificial intelligence).

While mechanical program understanding is unreasonable to expect, progress is occurring to automate much of the program manipulation now done by hand. Program transformations, adaptations, etc., are being developed that respect the behavior of the programs being manipulated. Formally, program behavior is *semantics*, and recent rapid progress in partial evaluation owes to advances in understanding the operational aspects of semantics.

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 non-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 is always 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. In other words, does one need *binding-time improvements*: transformations that do not change the algorithm's semantics, but make it easier for the specializer to separate binding-times?

Everchanging languages and systems. Unfortunately for many automation attempts, the ground is constantly shifting under one's feet in both software and in hardware (e.g., new architectures and changing preferences for imperative, logic, functional, and object-oriented programming). An ever-changing context makes systematization and automation quite hard; a well-known example is that methods good for yesterday's optimizing compilers are sometimes disastrous on today's machines.

Nonetheless, frequent change is in no way a good excuse for neglecting the application of automation in our own workplaces (indeed, this is embarrassing, given the enormous benefits given by computers in automating work done in other scientific and industrial fields!).

9. HISTORY

Jones et al. [1993] describe several approaches to and systems for partial evaluation. For an extensive bibliography including references to papers in Russian, see Sestoft and Zamulin [1988]. It is still being updated and is electronically available by anonymous ftp from file `pub/diku/dists/jones-book/partial-eval.bib.Z`.

Another source: `http://www.diku.dk/research-groups/topps/Bibliography.html`.

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 compiling may in principle be done by partial [Futamura 1971]. Turchin [1986], Ershov [1982] and Beckmann et al. [1976] realized the same independently in the mid-1970s and saw that even a compiler generator could be built by applying a partial evaluator to itself.

Practice. Lombardi and Raphael's [1964] papers on incremental computation were pathbreaking. In the mid-1970s a large partial evaluator was developed in Sweden for Lisp as used in practice (including imperative features and property lists) [Beckman et al. 1976] and a partial evaluator for Prolog [Komorowski 1982]. Trends to recognize partial evaluation as an important tool appeared among dedicated builders of compiler generators [Mosses 1979; Paulson 1984].

A wide range of languages has been covered in recent years, including first-order functional languages [Berlin and Weise 1990; Consel 1993; Jones et al. 1989; Romanenko 1988], higher-order languages including Scheme [Bondorf 1991; Gomard and Jones 1991b; Weise et al. 1991], typed languages [Launchbury 1991], logic programming including Prolog [Komorowski 1982; Lloyd and Shepherdson 1991; Safra and Shapiro 1986; Leuschel and De Schreye, 1995; Sahlin 1990] a term-rewriting language [Bondorf 1989], and imperative languages [Andersen 1994; 1992; Gomard and Jones 1991] including a subset of C.

Self-application. A nontrivial self-applicable partial evaluator that required handmade unfolding annotations for function calls was first developed in late 1984 and communicated in Jones et al. [1985]. Fully automatic self-applicable systems among the above are discussed in Andersen [1992], Bondorf [1990], Consel [1993], Gomard and Jones [1991a, 1991b], Jones et al. [1989] and Romanenko [1988].

New applications. An early motivation was optimization, so it is gratifying to see recent applications to scientific computing such as that of Berlin and Weise [1990] and Baier et al. [1994]. 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 [Nirkhe and Pugh 1992], incremental computation, debugging concurrent programs [Shapiro 1983], and parallel and pipelined computation [Pingali and Rogers 1990; Vasell 1993].

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.

10. 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,
—produce better results when specializing typed languages,
—avoid code explosion (by automatic means), and
—generate machine architectures tailor-made to the source language defined by an interpreter

Deeper-going and more advanced critical reviews may be found in Jones et al. [1993] and Ruf [1993].

ACKNOWLEDGMENTS

This article 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, Jacques Cohen, Robert Glück, John Launchbury, Patrick O'Keefe, Carolyn Talcott, Dan Weise, and the referees.

REFERENCES

ACM. 1991. *Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut. (*Sigplan Notices, 26*, 9, Sept.), ACM Press.

ACM. 1992. *Sigplan Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, (San Francisco).

ACM. 1993. *Sigplan Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (Copenhagen). ACM Press.

ACM 1994. *Sigplan Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, (Orlando, Florida) Univ. of Melbourne report 94/9, 1994.

ACM. 1995. *Sigplan Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, (San Diego, Calif.) ACM Press.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

ANDERSEN, L. O. 1994. Program analysis and specialization for the C programming language. DIKU, Dept. of Computer Science, Univ. of Copenhagen. DIKU Rep. No. 94/19.

ANDERSEN, L. O. 1992. C program specialization. *International Workshop on Compiler Construction,* (Paderborn, Germany), Springer-Verlag.

ANDERSEN, P. H. 1994. Partial evaluation applied to ray tracing. Res. Rep. DIKU, Dept. of Computer Science, Univ. of Copenhagen.

APPEL, A. 1988. Reopening closures. Unpublished report, Princeton Univ.

BAIER, R., GLÜCK, R., AND ZÖCHLING, R. 1994. Partial evaluation of numerical programs in Fortran. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Tech. Rep. 94/9, Univ. of Melbourne, Australia, 119–132.

BECKMAN, L., ET AL. 1976. A partial evaluator, and its use as a programming tool. *Artif. Intell. 7*, 4, 319–357.

BERLIN, A. AND WEISE, D. 1990. Compiling scientific code using partial evaluation. *IEEE Comput. 23*, 12 (Dec.) 25–37.

BJØRNER, D., ERSHOV, A. P., AND JONES, N. D. EDS. 1987. Partial evaluation and mixed computation. In *Proceedings of the IFIP TC2 Workshop* (Gammel Avernœs, Denmark, Oct.) North-Holland, 1988.

BONDORF, A. 1989. A self-applicable partial evaluator for term rewriting systems. In *TAPSOFT '89 Proceedings of the International Conference Theory and Practice of Software Development*, J. Diaz and F. Orejas, Eds. (Barcelona, Spain, Mar.) (Lecture Notes in Computer Science, 352), Springer-Verlag, 81–95.

BONDORF, A. 1990. *Self-applicable partial evaluation*. PhD thesis, DIKU, Univ. of Copenhagen. Revised version: DIKU Rep. 90/17.

BONDORF, A. 1991. Automatic autoprojection of higher order recursive equations. *Sci. Comput. Program. 17*, 3–34.

BURSTALL, R. M. AND DARLINGTON, J. 1977. A transformation system for developing recursive programs. *J ACM, 24*, 1 (Jan.) 44–67.

CONSEL, C. AND DANVY, O. 1989. Partial evaluation of pattern matching in strings. *Inf. Process. Lett., 30* (Jan.) 79–86.

CONSEL, C. AND NOEL, F. 1992. A general approach for run-time specialization and its application to C. In *ACM Symposium on Principles of Programming Languages*, (Orlando, Florida, Jan.) ACM Press.

CONSEL, C. AND DANVY, O. 1993. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*. ACM Press.

CONSEL, C. 1993. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 66–77.

DANVY, O. 1991. Semantics-directed compilation of non-linear patterns. *Inf. Process. Lett. 37*, 315–322.

ERSHOV, A. P. 1982. Mixed computation: Potential applications and problems for study. *Theor. Comput. Sci., 18*, 41–67.

ERSHOV, A. P., BJØRNER, D., FUTAMURA, Y., FURUKAWA, K., HARALDSON, A., AND SCHERLIS, W.

EDS. 1988. In *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation*, 1987. *New Generation Comput., 6*, 2,3. Ohmsha Ltd. and Springer-Verlag.

FUTAMURA, Y. 1971. Partial evaluation of computation process—an approach to a compiler-compiler. *Syst. Comput. Contr. 2*, 5, 45–50.

GOMARD, C. K. AND JONES, N. D. 1991a. Compiler generation by partial evaluation: a case study. *Structured Program. 12*, 123–144. Also as DIKU-report 88/24 and 90/16.

GOMARD, C. K. AND JONES, N. D. 1991b. A partial evaluator for the untyped lambda-calculus. *J. Funct. Program. 1* 1 (Jan.) 21–69.

HANNAN, J. AND MILLER, D. 1990. From operational semantics to abstract machines. In *1990 ACM Conference on Lisp and Functional Programming*, (Nice, France), ACM Press, June, 323–332.

HOLST, N. C. K. 1988. Language triplets: The AMIX approach. In *Partial Evaluation and Mixed Computation*, D. Bjørner, A. P. Ershov, and N. D. Jones, Eds., North-Holland, 167–185.

JACOBSEN, H. F. 1990. Speeding up the back-propagation algorithm by partial evaluation. DIKU Student Project 90-10-13, 32 pages. DIKU, Univ. of Copenhagen. (In Danish).

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

JONES, N. D. 1995. MIX ten years later. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 24–38.

JONES, N. D., GOMARD, C., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

JONES, N. D., SESTOFT, P., AND SØNDERGAARD, H. 1985. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications*, J.-P. Jouannaud, Ed. (Dijon, France) *Lecture Notes Computer Science, 202*, Springer-Verlag, 124–140.

JONES, N. D., SESTOFT, P., AND SØNDERGAARD, H. 1989. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp Symbolic Comput., 2*, 1, 9–50.

JØRGENSEN, J. 1992. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth ACM Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, Jan.) ACM, 258–268.

JØRRING, U. AND SCHERLIS, W. L. 1986. Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages* (St. Petersburg, Florida) 86–96.

KOMOROWSKI, H. J. 1982. 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) 255–267.

LAUNCHBURY, J. 1991. Projection factorisations in partial evaluation. Distinguished Dissertations in Computer Science. Cambridge Univ. Press.

LEONE, M. AND LEE, P. 1994. Lightweight run-time code generation. In *Proceedings of PEPM'94, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press.

LEUSCHEL, M. AND DE SCHREYE, D. 1995. Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, (La Jolla, California, June) ACM Press, 253–263.

LLOYD, J. W. AND SHEPHERDSON, J. C. 1991. Partial evaluation in logic programming. *J. Logic Program., 11*, 217–242.

LOMBARDI, L. A. AND RAPHAEL, B. 1964. Lisp as the language for an incremental computer. In *The Programming Language Lisp: Its Operation and Applications*, E. C. Berkeley and D. G. Bobrow, Eds. MIT Press, Cambridge, Massachusetts, 204–219.

MCCARTHY, J., ET AL. 1962. *LISP 1.5 Programmer's Manual*. MIT Computation Center and Research Laboratory of Electronics.

MOGENSEN, T. 1986. The application of partial evaluation to ray-tracing. Master's thesis, DIKU, Univ. of Copenhagen, Denmark.

MOSSES, P. 1979. SIS—semantics implementation system, reference manual and user guide. DAIMI Rep. MD-30, DAIMI, Univ. of Aarhus, Denmark.

NIRKHE, V. AND PUGH, W. 1992. 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, Jan.) ACM, 269–280.

PAGAN, F. G. 1991. *Partial Computation and the Construction of Language Processors*. Prentice-Hall, 166.

PAULSON, L. 1984. Compiler generation from denotational semantics. In *Methods and Tools for Compiler Construction*, B. Lorho, Ed. Cambridge University Press, 219–250.

PINGALI, K. AND ROGERS, A. 1990. Compiler parallelization for a simple distributed memory-machine. In *International Conference on Parallel Programming*, (St. Charles, Illinois).

PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUYE, J., KETHANA, L., WALPOLE, J., AND

ZHANG, K. 1995. Optimistic incremental specialization: streamlining a commercial operating system. In *ACM Symposium on Operating Systems Principles*.

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

RUF, E. 1993. Topics in online partial evaluation, Ph.D. thesis, Stanford Univ., California, Published as Tech. Rep. CSL-TR-93-563.

SAFRA, S. AND SHAPIRO, E. 1986. Meta interpreters for real. In *Information Processing 86*, H.-J. Kugler, Ed. North-Holland, 271–278.

SAHLIN, D. 1990. The Mixtus approach to automatic partial evaluation of full Prolog. In *Logic Programming: Proceedings of the 1990 North American Conference* (Austin, Texas, Oct.), S. Debray and M. Hermenegildo, Eds. MIT Press, 377–398.

SESTOFT, P. AND ZAMULIN, A. V. 1988. Annotated bibliography on partial evaluation and mixed computation. In *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation*. Ohmsha Ltd. and Springer-Verlag, 309–354.

SHAPIRO, E. 1983. *Algorithmic Program Debugging*. MIT Press, 1983.

Received February 1996; accepted June 1996

SØRENSEN, H., GLÜCK, R., AND JONES, N. D. 1994. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *European Symposium on Programming* (Glasgow) J.-P. Jouannaud, Ed., (Lecture Notes in Computer Science). Springer-Verlag.

SPERBER, M. AND THIEMANN, P. 1996. Realistic compilation by partial evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI'96*.

SPERBER, M. AND THIEMANN, P. 1995. The essence of LR parsing. In *ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation* (San Diego, California), ACM Press, 146–155.

TURCHIN, V. F. 1986. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst., 8*, 3 (July) 292–325.

VASELL, J. 1993. A partial evaluator for data flow graphs. In *ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation*. (Copenhagen) 206–215.

WEISE, D., CONYBEARE, R., RUF, E., AND SELIGMAN, S. 1991. Automatic online partial evaluation. In *Functional Programming Languages and Computer Architecture* (Cambridge, Massachusetts, Aug.) J. Hughes, Ed., (Lecture Notes in Computer Science, 523), ACM, Springer-Verlag, 165–191.