Partial evaluation

An article for Encyclopedia of Computer Science and Technology

Version 1.01 of 1996-04-09

Torben Æ. Mogensen[1] and Peter Sestoft[2]

1 Introduction: What is partial evaluation?

Partial evaluation is a technique to partially execute a program, when only some of its input data are available. Consider a program p requiring two inputs, x_1 and x_2. When specific values d_1 and d_2 are given for the two inputs, we can run the program, producing a result. When only one input value d_1 is given, we cannot run p, but can *partially evaluate* it, producing a version p_{d_1} of p specialized for the case where $x_1 = d_1$. Partial evaluation is an instance of *program specialization*, and the specialized version p_{d_1} of p is called a residual program.

For an example, consider the following C function `power(n, x)`, which computes `x` raised to the `n`'th power.

```
power(n, x)
int n,x;
{ int p;
  p = 1;
  while (n > 0) {
    if (n % 2 == 0) { x = x * x; n = n / 2; }
    else { p = p * x; n = n - 1; }
  }
  return(p);
}
```

Given values $n = 5$ and $x = 7$, we can compute `power(5,7)`, obtaining the result $7^5 = 16807$. (The algorithm exploits that $x^n = (x^2)^{n/2}$ for even integers n).

Suppose we need to compute power(n, x) for $n = 5$ and a great many different values of x. Then we can partially evaluate the function for $n = 5$, obtaining the following residual function:

```
power_5(x)
int x;
{ int p;
  p = x;
  x = x * x;
  x = x * x;
  p = p * x;
  return(p);
}
```

We can compute `power_5(7)` to obtain the result $7^5 = 16807$. In fact, for any input x, computing power_5(x) will produce the same result as computing

[1]DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK–2100 Copenhagen Ø, Denmark

[2]Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark

power(5,x). Since the value of variable n is available for partial evaluation, we say that n is *static*; conversely, the variable x is *dynamic* because its value is unavailable.

This example shows the strengths of partial evaluation: in the residual program power_5, all tests and all arithmetic operations involving n have been eliminated. The flow of control (that is, the conditions in the while and if statements) in the original program was completely determined by the static variable n.

Now suppose we needed to compute power(n,7) for many different values of n. This is the opposite problem of the above: now n is dynamic (unknown) and x is static (known). There is little we can do in this case, since the flow of control is determined by the dynamic variable n. One could imagine creating a table of precomputed values of 7^n for some values of n, but how are we to know which values are relevant?

In many cases some of the control flow is determined by static variables, and in these cases substantial speed-ups can be achieved by partial evaluation.

1.1 Notation

We can consider a program in two ways: as a function transforming inputs to outputs, and also as a data object, being input to or output from other programs. We need to distinguish the *function computed by a program* from the *program text* itself.

Writing p for the program text, we write $[\![\, p \,]\!]$ for the function computed by p, or $[\![\, p \,]\!]_L$ when we want to make explicit the language L in which p is written. Consequently, $[\![\, p \,]\!]_L\, d$ denotes the result of running program p with input d on an L-machine.

Now we can assert that power_5 is a correct residual program (in C) for power and given input $n = 5$:

$$[\![\, \texttt{power} \,]\!]_C\, (5, x) = [\![\, \texttt{power\_5} \,]\!]_C\, x$$

1.2 Interpreters and compilers

An interpreter *Sint* for language S, written in language L, satisfies for any S-program s and input data d:

$$[\![\, s \,]\!]_S\, d = [\![\, Sint \,]\!]_L [s, d]$$

That is, running s with input d on an S-machine gives the same result as using the interpreter *Sint* to run s on an L-machine. This includes possible nontermination of both sides.

A compiler *STcomp* for source language S, generating code in target language T, and written in language L, satisfies

$$[\![\, STcomp \,]\!]_L\, p = p' \;\text{ implies }\; [\![\, p' \,]\!]_T\, d = [\![\, p \,]\!]_S\, d \;\text{ for all } d$$

That is, p can be compiled to a target program p' such that running p' on a T-machine with input d gives the same result as running p with input d on an S-machine. Though the equation doesn't specify this, we normally assume compilation to always produce a target program.

2 Partial evaluators

2.1 What is a partial evaluator?

A partial evaluator is a program which performs partial evaluation. That is, it can produce a residual program by specializing a given program with respect to part of its input.

Let p be an L-program requiring two inputs x_1 and x_2 as above. A *residual program* for p with respect to $x_1 = d_1$ is a program p_{d_1} such that for all values d_2 of the remaining input,

$$[\![p_{d_1}]\!]\, d_2 = [\![p]\!]\, [d_1, d_2]$$

A *partial evaluator* is a program *peval* which, given a program p and a part d_1 of its input, produces a residual program p_{d_1}. In other words, a partial evaluator *peval* must satisfy:

$$[\![\mathit{peval}]\!][p, d_1] = p_{d_1} \quad \text{implies} \quad [\![p_{d_1}]\!]\, d_2 = [\![p]\!]\, [d_1, d_2] \quad \text{for all } d_2$$

This is the so-called *partial evaluation equation*, which reads as follows: If partial evaluation of p with respect to d_1 produces a residual program p_{d_1}, then running p_{d_1} with input d_2 gives the same result as running program p with input $[d_1, d_2]$. As for compilers, the equation does not guarantee termination of the left-hand side of the implication. In contrast to compilers we will, however, not always assume partial evaluation to succeed. While it is desirable for partial evaluation to always terminate, this is not guaranteed by a large number of existing partial evaluators. See sections 2.2 and 5.5 for more about the termination issue.

Above we have not specified the language L in which the partial evaluator is written, the language S of the source programs it accepts, or the language T of the residual programs it produces. These languages may be all different, but for notational simplicity we assume they are the same, $L = S = T$. Note that $L = S$ opens the possibility of applying the partial evaluator to itself (see below).

For an instance of the partial evaluation equation, consider $p =$ `power` and $d_1 = 5$, then from $[\![\mathit{peval}]\!][\texttt{power}, 5] = \texttt{power\_5}$ it must follow that `power(5,7) = power_5(7) = 16807`.

2.2 What is achieved by partial evaluation?

The definition of a partial evaluator does not stipulate that the specialized program must be any better than the original program. Indeed, it is easy to write a program *peval* which satisfies the partial evaluation equation in a trivial way, by prepending a new 'specialized' function `power_5` to the original program. The new function simply calls the original one with the given argument:

```
power(n, x)
int n,x;
{ int p;
  p = 1;
  while (n > 0) {
    if (n % 2 == 0) { x = x * x; n = n / 2; }
    else { p = p * x; n = n - 1; }
  }
  return(p);
}

power_5(x)
int x;
{ return(power(5, x)); }
```

While this program is a correct residual program, it is no faster than the original program, and quite possibly slower. Even so, the construction above can be used to prove existence of partial evaluators, a proof similar to Kleene's (1952) proof of the s-m-n theorem [63], a theorem that essentially stipulates the existence of partial evaluators in recursive function theory.

But, as the example in the introduction demonstrated, it is sometimes possible to obtain residual programs that are arguably faster than the original program. The amount of improvement depends both on the partial evaluator and the program being specialized. Some programs do not lend themselves to specialization, as no computation can be done before all input is known. Sometimes choosing a different algorithm may help, but in other cases the problem itself is ill-suited for specialization. An example is specializing the `power` function to a known value of `x`, as discussed in the introduction.

Looking at the definition of `power`, one would think that specialization with respect to a value of `x` would give a good result: the assignments, `p = 1;`, `x = x * x;` and `p = p * x;` do not involve `n`, and as such can be executed during specialization. The loop is, however, controlled by `n`. Since the termination condition is not known, we cannot fully eliminate the loop. But `x` and `p` will have different values in different iterations of the loop, so we cannot replace them by constants. Hence, we find that we cannot perform the computations on `x` and `p` anyway. We could force unfolding of the loop to keep the values of `x` and `p` known, but since there is no bound on the number of different values `x` and `p` can obtain, no finite amount of unfolding can eliminate `x` and `p` from the program.

This conflict between termination of specialization and quality of residual program is common. The partial evaluator must try to find a balance that ensures termination often enough to be interesting (preferably always) while yielding sufficient speed-up to be worthwhile. Due to the undecidability of the halting problem, no perfect strategy exists, so a suitable compromise must be found. See Section 5.5 for more on this subject.

3 Another approach to program specialization

A *generating extension* of a two-input program p is a program p_{gen} which, given a value d_1 for the first input of p, produces a residual program p_{d_1} for p with respect to d_1. In other words,

$$[\![p_{gen}]\!]\, d_1 = p_{d_1} \quad \text{implies} \quad [\![p]\!]\, [d_1, d_2] = [\![p_{d_1}]\!]\, d_2$$

The generating extension takes a given value d_1 of the first input parameter x_1 and constructs a version of p specialized for $x_1 = d_1$.

As an example, we show below a generating extension of the `power` program from the introduction:

```
power_gen(n)
int n;
{
  printf("{power_%d(x)\n",n);
  printf("int x;\n");
  printf("{ int p;\n");
  printf("  p = 1;\n");
  while (n > 0) {
    if (n % 2 == 0) { printf("  x = x * x;\n"); n = n / 2; }
    else { printf("  p = p * x;\n"); n = n - 1; }
  }
  printf("  return(p);\n");
  printf("}\n");
}
```

Note that `power_gen` closely resembles `power`: those parts of `power` that depend only on the static input `n` are copied directly into `power_gen`, and the parts that also depend on `x` are made into strings, which are printed as part of the residual program. Running `power_gen` with input $n = 5$ yields the following residual program:

```
power_5(x)
int x;
{ int p;
  p = 1;
  p = p * x;
  x = x * x;
  x = x * x;
  p = p * x;
  return(p);
}
```

This is almost the same as the one shown in the introduction. The difference is because we have now made an *a priori* distinction between static variables (`n`) and dynamic variables (`x`, `p`). Since `p` is dynamic, all assignments to it are made part of the residual program, even `p = 1`, which was executed at specialization time in the example shown in the introduction.

Later we shall see that a generating extension can be constructed by applying a sufficiently powerful partial evaluator to itself. One can even construct a generator of generating extensions that way.

4 Partial evaluation, interpreters, and compilation

4.1 Compilation using a partial evaluator

In Section 1.2 we defined an interpreter as a program taking two inputs: a program to be interpreted and input to that program:

$$[\![\, s \,]\!]_S\, d = [\![\, Sint \,]\!]_L [s, d]$$

We often expect to run the same program repeatedly on different inputs. Hence, it is natural to partially evaluate the interpreter with respect to a fixed, known program and unknown input to that program. Using the partial evaluation equation we get

$$[\![\, peval \,]\!][Sint, s] = Sint_s \;\text{ implies }\; [\![\, Sint_s \,]\!] d = [\![\, Sint \,]\!]_L [s, d] \;\text{ for all } d$$

Using the definition of the interpreter we get

$$[\![\, Sint_s \,]\!] d = [\![\, p \,]\!]_S\, d \;\text{ for all } d$$

The residual program is thus equivalent to the source program. The difference is the language in which the residual program is written. If the input and output languages of the partial evaluator are identical, then the residual program is written in the same language L as the interpreter $Sint$. Hence we have compiled s from S, the language that the interpreter interprets, to L, the language in which it is written.

4.2 Compiler generation using a self-applicable partial evaluator

We have seen that we can compile programs by partially evaluating an interpreter. Typically we will want to compile many different programs, which amounts to partially evaluating the same interpreter repeatedly with respect to different programs. This situation calls for optimization by yet another application of partial evaluation. Hence we use a partial evaluator to specialize a partial evaluator $peval$ with respect to a program $Sint$, but without the argument s of $Sint$. Using the partial evaluation equation we get:

$$\begin{aligned} &[\![\, peval \,]\!][peval, Sint] = peval_{Sint} \;\text{ implies} \\ &\qquad [\![\, peval_{Sint} \,]\!] s = [\![\, peval \,]\!][Sint, s] \;\text{ for all } s \end{aligned}$$

Using the results from above, we get

$$[\![\, peval_{Sint} \,]\!] s = Sint_s \;\text{ for all } s$$

for which we have

$$[\![\, Sint_s \,]\!] d = [\![\, s \,]\!]_S\, d \;\text{ for all } d$$

We recall the definition of a compiler from Section 1.2:

$$[\![\, STcomp \,]\!]_L\, p = p' \;\text{ implies }\; [\![\, p' \,]\!]_T\, d = [\![\, p \,]\!]_S\, d \;\text{ for all } d$$

We see that $peval_{Sint}$ fulfills the requirements for being a compiler from S to T. In the case where the input and output languages of the partial evaluator are identical, the language in which the compiler is written and the target language of the compiler are both the same as the language L in which the interpreter is written. Note that we have no guarantee that partial evaluation terminates, neither when producing the compiler nor when using it. Experience has shown that while this may be a problem, it is often the case that if compilation terminates for a few general programs then it terminates for all.

Note that the compiler $peval_{Sint}$ is a generating extension of the interpreter *Sint*, according to the definition shown in section 3. This generalizes to any program, not just interpreters: partially evaluating a partial evaluator *peval* with respect to a program p yields a generating extension $p_{gen} = peval_p$ for this program.

4.3 Compiler generator generation

Having seen that it is interesting to partially evaluate a partial evaluator, we may want to do this repeatedly: to partially evaluate a partial evaluator with respect to a range of different programs (e.g., interpreters). Again, we may exploit partial evaluation:

$$[\![peval]\!][peval, peval] = peval_{peval} \text{ implies } [\![peval_{peval}]\!]p = [\![peval]\!][peval, p] \text{ for all } p$$

Since $[\![peval]\!][peval, p] = peval_p$, which is a generating extension of p, we can see that $peval_{peval}$ is a *generator of generating extensions*. The program $peval_{peval}$ is itself a generating extension of the partial evaluator: $peval_{gen} = peval_{peval}$. In the case where p is an interpreter, the generating extension p_{gen} is a compiler. Hence, $peval_{gen}$ is a compiler generator, capable of producing a compiler from an interpreter.

4.4 Summary: The Futamura projections

Instances of the partial evaluation equation applied to interpreters, directly or through self-application of a partial evaluator, are collectively called the *Futamura projections*. The three Futamura projections are:

The first Futamura projection: compilation

$$[\![peval]\!][interpreter, source] = target$$

The second Futamura projection: compiler generation

$$[\![peval]\!][peval, interpreter] = compiler$$

$$[\![compiler]\!]source = target$$

The third Futamura projection: compiler generator generation

$$[\![peval]\!][peval, peval] = compilergenerator$$

$$[\![compilergenerator]\!]interpreter = compiler$$

The first and second equations were devised by Futamura in 1971 [40], and the latter independently by Beckman et al. [12] and Turchin et al. [94] around 1975.

5 Techniques for partial evaluation

5.1 Polyvariant specialization

Polyvariant specialization is a technique for partial evaluation which works for a range of languages. A program is thought of as a collection of *program points*, connected by *control-flow edges*. In a flow-chart language, program points and control-flow edges are, respectively, *labelled basic blocks* and *jumps*; in a functional language, they are *defined functions* and *function calls*; in a logic language, they are *predicates* and *predicate applications* (atoms).

Polyvariant specialization constructs a residual program by creating zero or more specialized variants of each program point, and connecting them by residual control-flow edges.

5.1.1 The exponentiation example revisited

To illustrate polyvariant specialization, consider the `power` function from Section 1 in flowchart form, with explicitly labelled basic blocks:

```
        p = 1
lab1:   if (n <= 0) goto lab3
        if (n % 2 != 0) goto lab2
        x = x * x
        n = n / 2
        goto lab1
lab2:   p = p * x
        n = n - 1
        goto lab1
lab3:   return(p)
```

A program on this form is specialized to given values of the static variables by specializing the basic blocks. For each basic block, and for each set of static variable values with which it may be executed, one creates a specialized basic block in the residual program. This is *polyvariant specialization* [24, 57].

For instance, the basic block labelled `lab1` may be executed with static $n = 5, p = 1$. Hence one creates a specialized basic block, whose label $\texttt{lab1}_{\{n=5,p=1\}}$ consists of the original label and bindings for the static variables. The body of the specialized basic block consists of the specialized residual commands from the original basic block. Naturally, the specialized version of a jump `goto lab` is itself a jump $\texttt{goto lab}_{\{\ldots\}}$ to a specialized (decorated) version of `lab`.

To see how this works, let us specialize the above program with the known value $n = 5$ and an unknown value for x. First we get to `lab1` with $n = 5, p = 1$. Using this information to specialize that basic block, we perform the conditional `if`s statically, find that `(n <= 0)` is false and `(n % 2 != 0)` is true, and so must jump to `lab2`, still with $n = 5, p = 1$. We create a residual `goto` command, and a new specialized label $\texttt{lab2}_{\{n=5,p=1\}}$:

$$\begin{array}{ll}\texttt{lab1}_{\{n=5,p=1\}}\texttt{:} & \\ & \texttt{goto lab2}_{\{n=5,p=1\}}\end{array}$$

The corresponding specialized basic block is the block at `lab2` specialized with respect to $n = 5, p = 1$. The assignment `p = p * x` specializes to the residual code `p = 1 * x`, since x is dynamic. This means that p is no longer static. The assignment `n = n - 1` can be executed because n is static. The new static environment has $n = 4$. Hence the `goto lab1` specializes to `goto` $\texttt{lab1}_{\{n=4\}}$ and we get:

```
lab2{n=5,p=1}:
        p = 1 * x
        goto lab1{n=4}
```

Note that we had to generate a constant expression `1` to represent the static value 1 of p in the residual program. We say that the static value of p has been *lifted* to appear in the residual program.

Next we must specialize the basic block at `lab1` with respect to $n = 4$, and so on. This process continues until specialized basic blocks have been created for all specialized labels occurring in the residual program. In total, the following residual program is obtained:

```
lab1{n=5,p=1}:
        goto lab2{n=5,p=1}
lab2{n=5,p=1}:
        p = 1 * x
        goto lab1{n=4}
lab1{n=4}:
        x = x * x
        goto lab1{n=2}
lab1{n=2}:
        x = x * x
        goto lab1{n=1}
lab1{n=1}:
        goto lab2{n=1}
lab2{n=1}:
        p = p * x
        goto lab1{n=0}
lab1{n=0}:
        goto lab3{n=0}
lab3{n=0}:
        return(p)
```

This can be simplified by replacing jumps (`goto`s) with the code they jump to; this is called *transition compression* or *unfolding*. The result is almost as in Section 1:

```
        p = 1 * x
        x = x * x
        x = x * x
        p = p * x
        return(p)
```

The technique of polyvariant specialization turns out to work for other languages too; this is demonstrated in later sections for functional languages and logic languages.

The specialization process builds a graph whose nodes are specialized program points (labels), and whose edges are residual control-flow edges (jumps).

This may be done by maintaining a set *pending* of the specialized program points still to be created, and a mapping *out* from specialized program points to specialized program code fragments (basic blocks). One repeatedly chooses and removes a program point *pp* from *pending*, constructs the corresponding specialized program code fragment $code_{pp}$, and extends the mapping *out* with $[pp \mapsto code_{pp}]$. Moreover, one extends the set *pending* by any new specialized labels reachable from $code_{pp}$. More precisely, *pending* is extended with the set $successors(pp) \setminus dom(out)$, where $successors(pp)$ is the set of program points pp' to which there is a jump `goto` pp' from $code_{pp}$.

To begin with, *pending* contains just the program's entry point together with the initial values of its static variables, and *out* is empty. The procedure terminates if and when *pending* becomes empty, in which case *out* contains the residual program. This process may fail to terminate, as discussed in Section 5.5 below.

5.2 Online versus offline partial evaluation

There are two types of partial evaluators. An *online* partial evaluator is a kind of generalized interpreter, which needs no *a priori* division of variables into static and dynamic. During partial evaluation, the environment maps static variables to concrete values, and dynamic variables to symbolic expressions. When processing an expression e, the partial evaluator makes an online decision whether to evaluate it (giving a concrete value), or to residualize it (giving a residual expression), based on the current bindings of the variables appearing in e.

An *offline* partial evaluator, by constrast, works in two phases. The first phase is a *binding-time analysis*, which classifies the program's variables into (definitely) static and (possibly) dynamic, and similarly classifies all operations. The second phase is the specialization proper. This phase simply uses the static/dynamic classification of variables and operations when processing an expression; all evaluate/residualize decisions have been made offline. It never uses the actual value of a variable or expression, unless the binding-time analysis guarantees that it is static and hence indeed is a concrete value.

Since offline partial evaluators rely on a program analysis, they are usually more conservative than online partial evaluators, missing some opportunities for specialization. On the other hand, offline specializers have a simpler structure, and may exploit the global knowledge about the program gained by the binding-time analysis. Experience shows that it is harder to construct self-applicable online specializers than offline ones.

Hybrids of online and offline specializers have been constructed. For instance, one may use a three-valued binding-time analysis, which classifies variables and expressions as 'definitely static', 'definitely dynamic', or 'undecided' [92]. The specialization phase will just obey the static and dynamic annotations, but use the actual (specialization time) value of variables to decide whether to evaluate or residualize.

A generator of generating extensions is similar to an offline partial evaluator, since a generating extension embodies an *a priori* distinction between early (static) inputs and late (dynamic) inputs. A generator of generating extension usually includes a binding-time analysis.

Online partial evaluation has been studied for Scheme by Ruf and Weise [82, 100] and by many researchers in the logic programming community.

5.3 Binding-time analysis

The classification of variables into static and dynamic is called a *division*. The division must be *congruent*: if the value of some dynamic expression e may be assigned to a variable y, then y must be made dynamic. The expression e is static if it contains no dynamic variables.

Considering again the flow-chart version of the `power` function in Section 5.1, we see that if n is static and x is dynamic from the outset, then p must be classified as dynamic because $p * x$ is assigned to p, whereas n remains static: n is never assigned a dynamic value.

A simple binding time analysis may be performed by means of an abstract interpretation in which each variable and expression takes one of the abstract values S (for static) or D (for dynamic). One builds an initial division in which all variables are S, except for the dynamic input parameters. Now all assignments in the program are abstractly executed, possibly reclassifying variables as dynamic to satisfy the congruence requirement, until no more variables need to be reclassified as dynamic.

Alternatively, binding-time analysis may be done by type inference with subtypes, where S is considered a subtype of D, meaning that S may be coerced to D (corresponding to the lifting of a static value). This kind of binding-time analysis may be implemented efficiently by constraint solving [53].

When composite data structures (tuples, records, lists) are considered, a data structure may be *partially static*. For instance, the value of a variable may be a pair whose first component is static, and whose second component is dynamic. This may be described by the binding-time $S \times D$. Similarly, a list of such pairs may be described by the binding-time $(S \times D)$ *list*. The type inference approach to binding-time analysis is especially useful for handling partially static data structures in strongly typed languages, such as Standard ML, Pascal, or C. Latently typed languages, such as Scheme, are handled essentially by considering dynamic expressions to be untyped.

When a division has been computed by the binding-time analysis, one must decide for each operation in the program whether it must be evaluated or residualized (producing residual code) during partial evaluation. An arithmetic operation must be residualized unless all its operands are static. An `if` statement must be residualized unless the condition is static. We shall assume that an assignment will be residualized unless the assigned variable is static. We also assume that all `goto`s are residualized (any excess `goto`s may be removed by subsequent transition compression).

To visualize the classification of operations, we annotate the dynamic operations by underlining. For the `power` function, the annotation would be:

```
        p = 1
lab1:   if (n <= 0) goto lab3
        if (n % 2 != 0) goto lab2
        x = x * x
        n = n / 2
        goto lab1
lab2:   p = p * x
        n = n - 1
        goto lab1
lab3:   return(p)
```

Doing polyvariant specialization of this program with $n = 5$, but blindly following the annotations, we obtain (after transition compression):

```
p = 1
p = p * x
x = x * x
x = x * x
p = p * x
return(p)
```

which is just the result obtained by the generating extension in Section 3. This is because the generator of generating extensions presuppose the *a priori* distinction between the static (n) and dynamic (x, p) variables.

5.4 Residual programs containing loops

The residual program generated above contains no loops; all conditionals were statically decidable and all transitions could be compressed. However, the machinery in Section 5.1 suffices for creating residual programs containing loops. Consider the following contrived example:

```
while (n > 0) {
   if (k == 0)
      { sum = sum + n * n; }
   else
      { sum = sum + k * (n - k); }
   n = n - 1;
  }
  return(sum);
```

Written as a flow-chart, the program is

```
lab1:   if (n <= 0) goto lab4
        if (k != 0) goto lab2
        sum = sum + n * n
        goto lab3
lab2:   sum = sum + k * (n - k)
        goto lab3
lab3:   n = n - 1
        goto lab1
lab4:   return(sum)
```

Let us specialize it with respect to static $k = 3$ and dynamic *sum* and n. Specializing the basic block at `lab1` with respect to $k = 3$ we must create a residual version of the first conditional, because n is dynamic, whereas the second conditional can be reduced, because k is static and non-zero, giving a residual jump to $\texttt{lab2}_{\{k=3\}}$:

```
lab1{k=3}:
        if (n <= 0) goto lab4{k=3}
        goto lab2{k=3}
```

Next we specialize the code at label `lab2` with respect to $k = 3$, obtaining:

```
lab2{k=3}:
        sum = sum + 3 * (n - 3)
        goto lab3{k=3}
```

Continuing in this manner, we obtain this residual program:

```
lab1{k=3}:
        if (n <= 0) goto lab4{k=3}
        goto lab2{k=3}
lab2{k=3}:
        sum = sum + 3 * (n - 3)
        goto lab3{k=3}
lab3{k=3}:
        n = n - 1
        goto lab1{k=3}
lab4{k=3}:
        return(sum)
```

After transition compression, we get:

```
lab1{k=3}:
        if (n <= 0) goto lab4{k=3}
        sum = sum + 3 * (n - 3)
        n = n - 1
        goto lab1{k=3}
lab4{k=3}:
        return(sum)
```

The decorated labels $\texttt{lab1}_{\{k=3\}}$ and $\texttt{lab4}_{\{k=3\}}$ may be replaced by simple ones, such as `lab1r` and `lab4r`. Then we see that partial evaluation has eliminated the tests on k inside the loop; effectively, they were found to be loop-invariant. The loop is recreated in the residual program simply because the jump at $\texttt{lab3}_{\{k=3\}}$ goes back to the specialized program point $\texttt{lab1}_{\{k=3\}}$ at the beginning of the program.

5.5 Termination of partial evaluation

Transition compression should be applied with care in the program just shown. An attempt to (repeatedly) unfold all remaining occurrences of $\texttt{goto lab1}_{\{k=3\}}$ would never terminate. Infinite looping due to transition compression is avoided fairly easily; either by unfolding a jump to a (residual) label only if there is exactly one way to reach that label [21], or by ascertaining that unfolding must stop due to some descending chain condition [87].

Termination problems caused by *infinite specialization* are harder to deal with. For illustration, consider again the `power` program in Section 5.1, but now with static $x = 7$, static p, and dynamic n. A straightforward application of polyvariant specialization will attempt to produce an infinite residual program:

```
lab1{x=7,p=1}:
        if (n <= 0) goto lab3{x=7,p=1}
        if (n % 2 != 0) goto lab2{x=7,p=1}
        n = n / 2
        goto lab1{x=49,p=1}
lab3{x=7,p=1}:
        return(1)
lab2{x=7,p=1}:
        n = n - 1
        goto lab1{x=7,p=7}
lab1{x=49,p=1}:
        if (n <= 0) goto lab3{x=49,p=1}
        if (n % 2 != 0) goto lab2{x=49,p=1}
        n = n / 2
        goto lab1{x=2401,p=1}
lab3{x=49,p=1}:
        return(1)
lab2{x=49,p=1}:
        n = n - 1
        goto lab1{x=49,p=49}
        ...
```

This program is incomplete, and it cannot be completed using a finite number of program points, if we insist on keeping x and p static.

In an *online* partial evaluatior, one may recognize that the configuration $\{x = 49, p = 1\}$ is 'similar' to the previously encountered $\{x = 7, p = 1\}$ and that the two program points should therefore be merged into a single more general one, e.g. by making x dynamic (which eventually forces p to be dynamic also). This process is called *generalization*.

In an *offline* partial evaluator, one may recognize after binding-time analysis that the tests are dynamic (not decided by the static variables), and that static data are constructed under dynamic control. This is a sign of danger, indicating that x and p should be made dynamic too, making specialization completely trivial (but safe).

Holst developed a finiteness analysis and used it to ensure termination of polyvariant specialization [54].

5.6 Generalized partial evaluation

One more lesson may be learnt from the (partially constructed) residual program just shown. The basic block labelled `lab3`$_{\{x=49,p=1\}}$ is superfluous. Reaching it would require the tests `(n <= 0)` and `(n % 2 != 0)` to fail and the test `((n/2 <= 0)` to succeed, which is impossible for integral n; the former two imply that $n \geq 2$.

The superfluous basic block is created because the static environment (as outlined above) takes into account only the values of static variables (x and p), not the outcome of previously encountered dynamic tests (on n). Polyvariant specialization may be enhanced to do so, giving *generalized partial evaluation*. Then a theorem prover is required to decide static conditionals and to decide whether two static environments are equivalent [41]. In certain data domains and applications, less powerful methods may suffice [46].

6 Partial evaluation for other languages

6.1 Functional languages

6.1.1 First-order languages

Partial evaluation of a first-order functional language may be done by polyvariant specialization as described in Section 5.1 above. The notions of *label*, *basic block*, and *global variable* must be replaced by the notions of function name, function definition, and function parameter. Henceforth a specialized program point is a specialized function name, and a residual program is a collection of specialized function definitions.

For illustration, consider a functional version of the power program from Section 1, here using Standard ML syntax:

```
fun power(n, x) =
    if n = 0 then 1
    else if n mod 2 = 0 then power(n div 2, x * x)
                        else x * power(n-1, x)
```

Specializing the function power with respect to static $n = 5$ and dynamic x, we obtain

```
fun power{n=5}(x) = x * power{n=4}(x)
and power{n=4}(x) = power{n=2}(x * x)
and power{n=2}(x) = power{n=1}(x * x)
and power{n=1}(x) = x * power{n=0}(x)
and power{n=0}(x) = 1
```

Note that a specialized function name $\mathtt{power}_{\{n=5\}}$ consists of an original function name power together with a binding for the static parameters, here just n. The residual program may be simplified by unfolding trivial function calls (and reducing the subexpression (x * x) * 1 arising from this unfolding):

```
fun power{n=5}(x) = x * power{n=2}(x * x)
and power{n=2}(x) = x * x
```

This residual program is equivalent to that generated for the C version of power in Section 1.

Binding-time analysis may proceed as for a flow-chart language. For each application $(f\,e)$ where e may be dynamic, reclassify the formal parameter of f to dynamic. Since the language is first-order, f must be a known function.

As for flow-chart languages, a partial evaluator may either be offline or online. An offline partial evaluator will perform a binding-time analysis of the program, to classify all parameters as either static or dynamic, before embarking on the specialization phase proper. A complete description of a simple offline self-applicable partial evaluator for a first-order functional language may be found in [58, Chapter 5 and Appendix A].

6.1.2 Higher-order functional languages

Polyvariant specialization can be applied to higher-order functional languages (in which functions may be passed around as values) as well. The main new challenges are: how to represent static functional values during partial evaluation, how to lift functional values from static to dynamic, how to specialize with respect to functional values, and how to do binding-time analysis.

A functional value may be represented by a closure (g, vs) consisting of a function name g together with the values vs of the static free variables in g's body.

Lifting of a (partially) static functional value to a dynamic value is complicated and is usually avoided in offline partial evaluators, by requiring that every (partially) static functional value must be applied to an argument. Any functional value occurring in a dynamic context will be reclassified as dynamic by the binding-time analysis.

Specializing a function f with respect to a fully static functional closure (g, vs) is simple; just specialize with respect to the function name g and the values vs of the (static) free variables.

Specializing f with respect to a partially static (g, vs) is more involved, since the body of g may have dynamic free variables. These variables may be free also in the residual expression resulting from applying (g, vs). Hence the dynamic free variables must be lifted out of g's body at specialization time, and must be passed as extra parameters to the residual function $f_{(g,vs)}$.

A higher-order functional program may contain applications $(e_1\, e_2)$ where e_1 evaluates to some function. A closure analysis can provide an approximation to the set of functions that e_1 may evaluate to; using this information, binding-time analysis may proceed as for a first-order language [58, Chapter 15]. Alternatively, the binding-time analysis may be based on type inference [53]; this is preferable if one wants to permit partially static data structures also.

Self-applicable partial evaluators exist for realistic higher-order functional languages such as Scheme [20, 21, 28, 29, 31] and Standard ML [70] as well as for the call-by-value lambda calculus with some extensions [51], and for the pure lambda calculus [74, 75]. For further information, see e.g. [58, Chapter 10]; for full details, see the above-mentioned papers.

6.2 Logic programming languages (Prolog)

A distinguishing feature of Prolog and other logic programming languages is the ability to run with incomplete input. While this seems similar to partial evaluation, there are a number of differences:

- The result of running a Prolog program with incomplete input is a (possibly) infinite list of instantiations of both input and output variables. Though this can be considered a list of facts, and hence a restricted form of program, we generally want a partial evaluator to be able to produce non-trivial residual programs, possibly containing loops.
- Prolog has some non-logical features that means that running a program with incomplete input is not a generalization of running with complete input. As an example, calling the predicate defined by

```
p(X,Y) :- var(X), X = Y.
```

with partially instantiated input `p(A,a)` returns the result `A = a`, whereas running with complete input `p(a,a)` would fail.

Most of the research in partial evaluation (or *partial deduction*, as it is often called) of logic languages has tended to avoid the second issue by working with pure logic languages [43, 68]. Some systems, however, deal with non-logical features of Prolog [76, 84].

Partial evaluation of logic languages is typically done using the same basic techniques as for functional languages: call unfolding and polyvariant specialization, program points being predicates. A major source of speed-up in partially evaluating logic programs is the ability to detect failing computations at specialization time, and cut these away in the residual program. This way, not only static computations but also dynamic computations in failing branches can be eliminated by partial evaluation. This makes the potential speed-up by partial evaluation greater in logic languages than in functional or imperative languages.

Online specialization has been the preferred technique in the logic language community, usually combined with powerful techniques for avoiding non-termination [23, 71]. In logic languages, online specialization presents more opportunities for specialization than offline specialization, because unification will often instantiate otherwise dynamic variables. When self-application has been a major goal, offline specialization has been used also [52, 76].

An example of Prolog specialization is shown below. It specializes a program for regular expression matching

```
accepts(R,[])    :- nullable(R).
accepts(R,[C|S]) :- first(R,C), next(R,C,R1), accepts(R1,S).
```

The program takes a regular expression and a string as arguments. If the string is empty, the regular expression is tested for nullability (acceptance of empty string). If the string starts with a character `C`, it is tested whether this is among the `first` set of the regular expression. If this is the case, a new regular expression for matching the rest of the string is produced by `next`. The predicates `nullable`, `first` and `next` are not shown, but note that the set of `C`s for which `first` succeeds is determined by `R`. Hence, partial evaluation of `first` with respect to a known `R` and unknown `C` will yield a number of instantiations of `C`. Specializing the program above with respect to `R` being the regular expression $(a|b)^*aba$ yields the following residual program:

```
accepts_0([a|S]) :- accepts_1(S).
accepts_0([b|S]) :- accepts_0(S).

accepts_1([a|S]) :- accepts_1(S).
accepts_1([b|S]) :- accepts_2(S).

accepts_2([a|S]) :- accepts_3(S).
accepts_2([b|S]) :- accepts_0(S).

accepts_3([]).
accepts_3([a|S]) :- accepts_1(S).
accepts_3([b|S]) :- accepts_2(S).
```

Since `nullable` depends only on static values, it is completely eliminated, only visible as failed or true cases for the empty string. The call to `first` has instantiated C with a or b. This instantiation has made it possible to fully evaluate next, which has yielded a total of four different regular expressions, each giving rise to a specialized version of accepts. For accepts_0, the regular expression is $(a|b)^*aba$, for accepts_1 it is $(a|b)^*aba|ba$, for accepts_2 it is $(a|b)^*aba|a$, and for accepts_3 it is $(a|b)^*aba|ba|\epsilon$.

6.3 A full imperative language (C)

We have seen that polyvariant specialization suffices for partial evaluation of flow-chart languages, and hence for simple imperative languages. A realistic imperative language, such as C, includes composite data structures (records and indexed arrays), pointers and dynamic data structures, functions which may have side effects on global variables, etc.

An offline partial evaluator for C needs a sophisticated binding-time analysis to deal with pointers and composite data structures. For instance, a pointer variable p may be dynamic, or the pointer may be static but point to a dynamic object, or both the pointer and the pointed-to object may be static.

The binding-time analysis may require programs to be 'well-behaved'. Assume that a is an array, and that the program contains an assignment of the form a[n] = e, where e is dynamic. Then in principle any variable in the program may become dynamic as a result of this assignment, in case the address a[n] is outside the allocated array a. This would be too conservative, making partial evaluation trivial. Instead, one should require programs to be well-behaved, so that any such address is indeed inside a.

For a taste of the difficulties caused by the combination of non-local side effects and (recursive) functions, consider a function which has a side effect on a static global variable, but where the side effect is controlled by some dynamic expression *dyn*:

```
int global;
int main(...)
{ foo(...);
  stmts
}

void foo(...)
{ if dyn global = 1;
  else global = -1;
}
```

After the call to foo, the value of global may be either 1 or -1, but we cannot know which one at partial evaluation time, because *dyn* is dynamic. The simplest solution is to reclassify global as dynamic, but this wastes static information which might be useful when partially evaluating *stmts*. Another solution is to unfold the call to foo, giving a residual program of this form:

```
int global;
int main(...)
{ if dyn { stmts{global=1}; }
  else { stmts{global=-1}; }
}
```

Here $stmts_{\{global=1\}}$ is a specialized version of *stmts*. However, when function `foo` is recursive, such unfolding is impossible. A third solution is to introduce a (dynamic) continuation variable `cont`, which will be assigned a different value in each branch of the residual version `foo´` of `foo`. In function `main`, after the call to `foo´`, there will be a switch on `cont`:

```
int global;
int main(...)
{ foo´(...);
  switch (cont) {
     case 1: stmts{global=1};    break;
     case 2: stmts{global=-1};   break;
  }
}
```

However, this will not work when `foo` is recursive, and the recursion is under dynamic control, since the number of paths through `foo` will not be statically bounded in that case. Hence for recursive procedures, the only feasible option may be to reclassify `global` as dynamic.

These and many other problems were studied by Lars Ole Andersen, who constructed two systems for specialization of C programs. The first one is a self-applicable partial evaluator for a C subset, including procedures as well as pointers and arrays [6, 8]. The second one is a generator of generating extensions for all of ANSI C [9]; the latter system can be licensed from the University of Copenhagen.

The techniques for C should carry over to e.g. Ada, Modula, or Pascal with little modification, but to our knowledge this has not been done.

7 Partial evaluation in perspective

7.1 Program specialization without a partial evaluator

So far we have focused mainly on specialization using a partial evaluator. But the ideas and methods presented here can be, and indeed have been, used without using a partial evaluator.

Specialization by hand

It is quite common for programmers to hand-tune code for particular cases. Often this amounts to doing partial evaluation by hand. As an example, here is a quote from a paper [80] about the programming of a video-game:

> How Nevryon manages to keep up its speed [...] Basically there are two ways to write a routine:
> It can be one complex multi-purpose routine that does everything, but not quickly. For example, a sprite routine that can handle any size and flip the sprites horizontally and vertically in the same piece of code.
> Or you can have many simple routines each doing one thing. Using the sprite routine example, a routine to plot the sprite one way, another to plot it flipped vertically and so on.

> The second method means more code is required but the speed advantage is dramatic. Nevryon was written in this way and had about 20 separate sprite routines, each of which plotted sprites in slightly different ways.

Clearly specialization is used. But it is doubtful that a general purpose partial evaluator was used to do the specialization. Instead the specialization has been performed by hand, possibly without ever explicitly writing down the general purpose routine that forms the basis for the specialized routines.

Using hand-written generating extensions

We saw in Section 3 how a generating extension for the `power` function was easily produced from the original code, using knowledge about which variables contained values known at specialization time. While it is not always quite so simple as in this example, it is often not particularly difficult to write generating extensions of small to medium sized procedures or programs.

In situations where no partial evaluator is available, this is often a viable way to obtain specialized programs. Using a generating extension instead of writing the specialized versions by hand is useful when either a large number of variants must be generated, or when it is not known in advance what values the program will be specialized with respect to.

A common use of hand-written generating extensions is for run-time code generation, where a piece of specialized code is generated and executed, all at run-time. As in the sprite example above, one often generates specialized code for each plot operation when large bitmaps are involved. The typical situation is that a general purpose routine is used for plotting small bitmaps, but special code is generated for large bitmaps. The specialized routines can exploit knowledge about the alignment of the source bitmap and the destination area with respect to word boundaries, as well as clipping of the source bitmap. Other aspects such as scaling, differences in colour depth etc. have also been targets for run-time specialization of bitmap-plotting code.

Hand-written generating extensions have been used for optimizing parsers by specializing with respect to particular tables [78], and for converting interpreters into compilers [77].

Handwritten generating extension generators

In recent years, it has become popular to write a generating extension generator instead of a partial evaluator [9, 16, 55], but the approach itself is quite old [12].

A generating extension generator can be used instead of a traditional partial evaluator as follows. To specialize a program p with respect to data d, first produce a generating extension p_{gen}, then apply p_{gen} to d to produce a specialized program p_d.

Conversely, a self-applicable partial evaluator can produce a generating extension generator (cf. the third Futamura projection), so the two approaches seem equally powerful. So why write a generating extension generator instead of a self-applicable partial evaluator? Some reasons are:

- The generating extension generator can be written in another (higher level) language than the language it handles, whereas a self-applicable partial evaluator must be able to handle its own text.
- For this reason, among others, it may be easier to write a generating extension generator than a self-applicable partial evaluator.
- A partial evaluator must contain an interpreter, which may be problematic for typed languages, as explained below. Neither the generating extension generator, nor the generating extensions, need to contain an interpreter.

When writing an interpreter for a strongly typed language, one must use a single type in the interpreter to represent an unbounded number of types used in the programs that are interpreted. The same is true for a partial evaluator: a single universal type must be used for the static input to the program that will be specialized. Hence, the static input must be coded. This means that the partial evaluation equation must be modified to take this coding into account:

$$[\![\, peval \,]\!][p, \overline{d_1}] = p_{d_1} \wedge [\![\, p \,]\!]\, [d_1, d_2] = d' \text{ implies } [\![\, p_{d_1} \,]\!]\, d_2 = d'$$

where overlining means that a value is coded, e.g. $\overline{d_1}$ is the coding of the value of d_1.

When self-applying the partial evaluator, the static input is a program. The program is normally represented in a special data type that represents program text. This data type must now be coded in the universal type:

$$[\![\, peval \,]\!][peval, \overline{p}] = p_{gen} \text{ implies } [\![\, peval \,]\!]\, [p, \overline{d_1}][\![\, p_{gen} \,]\!]\, \overline{\overline{d_1}} = p_{d_1}$$

This double encoding is space- and time-consuming, and has been reported to make self-application intractable, unless special attention is paid to make the encoding compact [67]. A generating extension produced by self-application must also use the universal type to represent static input, even though this will always be of the same type.

This observation leads to the idea of making generating extensions accept uncoded static input. To achieve this, the generating extension generator simply copies the type declarations of the original program into the generating extension. The generating extension generator takes a single input: a program, and need not deal with arbitrarily typed data. A generating extension handles values from a single program, the types of which are known when the generating extension is constructed. Hence, neither the generator of generating extensions, nor the generating extensions themselves, need to handle arbitrarily typed values. The equation for specialization using a generating extension generator is shown below. Note the absence of coding.

$$[\![\, gengen \,]\!][p] = p_{gen} \wedge [\![\, p_{gen} \,]\!]\, d_1 = p_{d_1} \text{ implies } [\![\, p \,]\!]\, [d_1, d_2] = [\![\, p_{d_1} \,]\!]\, d_2$$

We will usually expect generator generation to terminate, but, as for normal partial evaluation, allow the construction of the residual program (performed by p_{gen}) to loop.

7.2 When is partial evaluation worthwhile?

In Section 2.2 we saw that we cannot always expect speed-up from partial evaluation. Sometimes no significant computations depend on the known input only, so virtually all the work is postponed until the residual program is executed. Even if computations appear to depend on the known input only, evaluating these during specialization may require infinite unfolding (as seen in Section 2.2) or just so much unfolding that the residual programs become intractably large.

On the other hand, the example in Section 1 manages to perform a significant part of the computation at specialization time. Even so, partial evaluation will only pay off if the residual program is executed often enough to amortize the cost of specialization.

So, we must have two conditions before we can expect any benefit from partial evaluation:

1) There are computations that depend only on static data.
2) These are executed repeatedly, either by repeated execution of the program as a whole, or by repetition (looping or recursion) within a single execution of the program.

The static (known) data can be obtained in several ways: it may be constants appearing in the program text or it can be part of the input.

It is quite common that library functions are called with some constant parameters, such as format strings, so in some cases partial evaluation may speed up programs even when no input is given. In such cases the partial evaluator works as a kind of optimizer, often achieving speed-up when most optimizing compilers would not. On the other hand, most partial evaluators may loop or create an excessive amount of code while trying to optimize programs, and hence are ill-suited as default optimizers.

Specialization with respect to partial input is the most common situation. Here there are often more opportunities for speed-up than just exploiting constant parameters. In some cases (e.g., when specializing interpreters) most of the computation can be done during partial evaluation, sometimes yielding speed-ups by an order of magnitude or more, similar to the speed difference between interpreted and compiled programs. When you have a choice between running a program interpreted or compiled, you will choose the former if the program is only executed a few times and contains no significant repetition, whereas you will want to compile it if it is run many times or involves much repetition. The same principle carries over to specialization.

Partial evaluation often gets most of its benefit from replication: loops are unrolled and the index variables exploited in constant folding, or functions are specialized with respect to several different static parameters. In some cases this replication can result in enormous residual programs, which may be undesirable even if much computation is saved. In the example in Section 1 the amount of unrolling and hence the size of the residual program is proportional to the logarithm of n, the static input. This expansion is small enough that it doesn't become a problem. If the expansion were linear in n, it would be acceptable for small values of n. Specialization of interpreters typically yield residual programs that are proportional to the size of the source program, which is reasonable. On the other hand, quadratic or exponential expansion is hardly ever acceptable.

It may be hard to predict the amount of replication caused by a partial evaluator. In fact, seemingly innocent changes to a program can dramatically change the expansion done by partial evaluation, or even make the difference between termination or nontermination of the specialization process. Similarly, small changes can make a large difference in the amount of computation that is performed during specialization and hence the speed-up obtained. This is similar to the way parallelizing compilers are sensitive to the way programs are written. Hence, specialization of off-the-shelf programs often require some (usually minor) modification to get optimal benefit from partial evaluation. Ideally, the programmer should write his program with partial evaluation in mind, avoiding the structures that can cause problems, just like programs for parallel machines are best written with the limitations of the compiler in mind.

7.3 Partial evaluation, optimizing compilers, and modern machines

Many compilers perform transformations such as constant folding and inlining (of small functions) to improve target programs. These transformations are similar to some of those performed by a partial evaluator. However, in contrast to a partial evaluator, a compiler rarely produces more than one specialized version of a given piece of code (except possibly by inlining). This kind of specialization is essential in partial evaluators, and must be handled correctly also in the presence of loops and recursive procedures.

With the complex memory hierarchies of modern computer hardware it is hard to know when a program modification actually achieves a speed-up. Exploiting the memory hierarchy well (data registers and instruction pipeline, two or more levels of cache, main memory, and virtual memory) is crucial for the performance of modern machines. Hence it may be detrimental to unroll a loop so that it does not fit in the cache, but beneficial to inline a procedure if this replaces indirect jumps by linear code sequences. How much unrolling, inlining, or replication to perform is machine dependent, and you often see optimizations that improve performance on one machine but degrade it on others.

With the increasing degree of micro-parallelism in modern microprocessors, one may even get no benefit from eliminating the static computations, as they may not be part of the critical path and hence may be executed in parallel with the dynamic computations. On the other hand, the elimination of variables by specialization reduces register pressure, and unrolling of loops and inlining of functions increase basic block size, giving more opportunities for low-level optimization.

This means that it is hard to predict the amount of speed-up obtained by partial evaluation. Examples exist where a residual program is twice as fast as the original program on one machine is, and slower than the original on another machine. The speed-up is also affected by the optimizations performed when compiling the residual programs.

8 Applications of partial evaluation

We saw in Section 4 that partial evaluation can be used to compile programs and to generate compilers. This has been one of the main practical uses of partial evaluation. Not for making compilers for C or similar languages, but for rapidly obtaining implementations of acceptable performance for experimental or special-purpose languages. Since the output of the partial evaluator typically is in a high-level language, a traditional compiler is used as a back-end for the compiler generated by partial evaluation [1, 14, 25, 27, 30, 33, 61]. In some cases, the compilation is from a language to itself. In this case the purpose is to make certain computation strategies explicit (e.g., continuation passing style) or to add extra information (e.g., for debugging) to the program [20, 42, 83, 93].

Many types of programs, e.g. scanners and parsers, use a table or other data structure to control the program. It is often possible to achieve speed-up by partially evaluating the table-driven program with respect to a particular table [7, 78]. However, this may produce very large residual programs, as tables (unless sparse) often represent the information more compactly than does code.

These are examples of converting *structural knowledge representation* to *procedural knowledge representation*. The choice between these two types of representation has usually been determined by the idea that structural information is compact and easy to modify but slow to use while procedural information is fast to use but hard to modify and less compact. Automatically converting structural knowledge to procedural knowledge can overcome the disadvantage of difficult modifiability of procedural knowledge, but retains the disadvantage of large space usage.

Section 7.1 mentioned a few applications of specialization to computer graphics. This has been one of the areas that have seen most applications of partial evaluation. An early example is [49], where an extended form of partial evaluation is used to specialize a renderer used in a flight simulator.

In a flight simulator the same landscape is viewed repeatedly from different angles. Though the occlusion of surfaces depend on the angle of view, it is often the case that the knowledge that a particular surface occludes (or not) another can decide the occlusion question of other pairs of surfaces. Hence, the partial evaluator simulates the sorting of surfaces and when it cannot decide which of two surfaces must be plotted first, it leaves that test in the residual program. Furthermore, it uses the inequalities of the occlusion test as positive and negative constraints in the branches of the conditional it generates. These constraints are then used to decide later occlusion tests (by attempting to solve the constraints by the Simplex method). Each time a test cannot be decided more information is added to the constraint set (which effectively constrains the view-angle), allowing more later tests to be decided. Goad reports that for a typical landscape with 1135 surfaces (forming a triangulation of the landscape) the typical depth of paths in the residual decision tree was 27, compared to the more than 10000 comparisons needed for a full sort [49]. This rather extreme speed-up is due to the nature of landscapes: many surfaces are almost parallel, and hence can occlude each other only in very narrow viewing angles.

Another graphics application has been ray-tracing. In ray-tracing, a scene is rendered by tracing rays (lines) from each pixel on the screen into an imaginary world behind the scene, testing which objects these rays hit. The process is repeated for all rays using the same fixed scene. Since there may be millions of pixels (and hence rays) in a typical ray-tracing application, specialization with respect to a fixed scene but unknown ray can give speed-up even for rendering single pictures. Speed-ups of more than 6 have been reported for a simple ray-tracer [73]. For a more realistic ray-tracer, speed-ups in the range 1.5 to 3 have been reported [10]. The speed-up is gained from several sources: the ray/object intersection routine is specialized for each object and the (highly parametrized) shading (colouring) function is specialized for each object. Furthermore, the representation of the scene is converted to procedural form.

Figure 1 is an example of a ray-traced picture made by the ray-tracer from [73]. The picture shows a 3D diagram of the process of partial evaluation: A program P and one of its inputs x are fed to the partial evaluator PE yielding a residual program P_x.

Partial evaluation has also been applied to numerical computation, in particular simulation programs. In such programs, part of the model will be constant during the simulation while other parts will change. By specializing with respect to the fixed parts of the model, some speed-up can be obtained. An example is the N-body problem, simulating the interaction of moving objects through gravitational forces. In this simulation, the masses of the objects are constant, whereas their position and velocity change. Specializing with respect to the mass of the objects can speed up the simulation. Berlin reports speed-ups of more than 30 for this problem [15]. However, the residual program is written in C whereas the original one was in Scheme, which may account for part of the speed-up. In another experiment, specialization of some standard numerical algorithms gave speed-ups ranging from none at all to about 5 [47].

When neural networks are trained, they are usually run several thousand times on a number of test cases. During this training, various parameters will be fixed, e.g. the topology of the net, the learning rate and the momentum. By specializing the trainer to these parameters, speed-ups of 25 to 50% are reported [56].

This list of applications is not exhaustive, but should give an impression of the range of possibilities.

9 Further reading

Here we first sketch the history from 1952 to 1984, then give a number of pointers to the literature on partial evaluation and some related topics. The book by Jones, Gomard, and Sestoft [58] includes more material on the subjects mentioned above, and a large bibliography; the updated source text for that bibliography is available for anonymous ftp from `ftp.diku.dk` as file `pub/diku/dists/jones-book/partial-eval.bib.Z`.

9.1 History

Kleene's s-m-n theorem (1952) asserts the feasibility of partial evaluation [63], and his constructive proof provides the design for a partial evaluator. This design did not, and was not intended to, provide any improvement of the specialized program. Such improvement, by symbolic reductions or similar, has been the goal in all subsequent work in partial evaluation.

Lombardi is probably the first one to use the term 'partial evaluation' [69]. Futamura is the first researcher to consider a partial evaluator as a *program* as well as a transformer, and thus to consider the application of the partial evaluator to itself [40]. Futamura's paper gives the equations for compilation and compiler generation by partial evaluation, but not for compiler generator generation. The

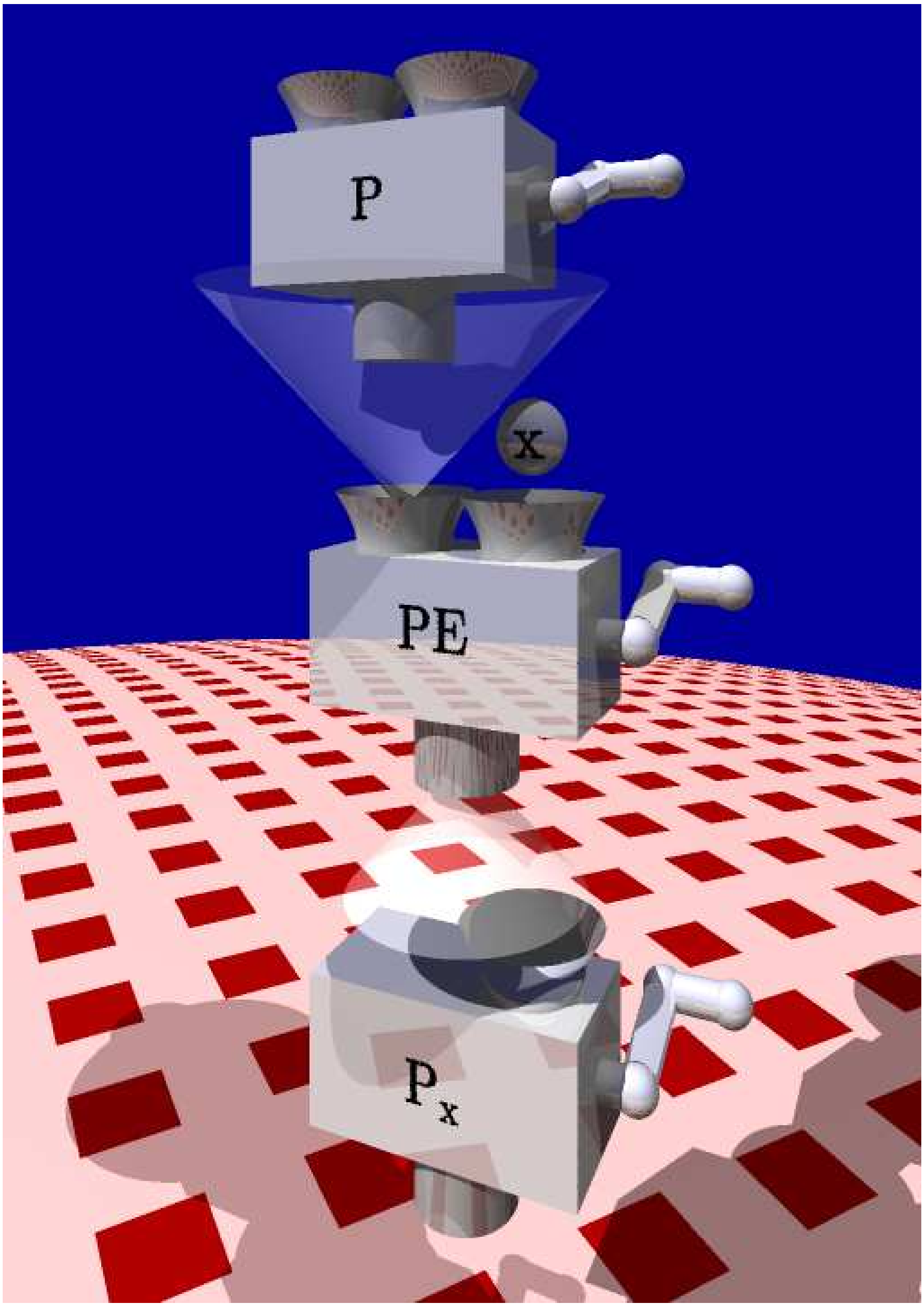

Figure 1: Partial evaluation in action

three equations were called the Futamura projections by Andrei Ershov [38]. Futamura's early ideas were not implemented.

Around 1975, Beckman, Haraldsson, Oskarsson, and Sandewall developed a partial evaluator called Redfun for a substantial subset of Lisp [12], and described the possibility of compiler generator generation by double self-application.

Turchin and his group also worked with partial evaluation in the early 1970s, in the context of the functional language Refal, and gave a description of self-application and double self-application [94]. The history of that work is briefly summarized in English in [95].

Andrei Ershov worked with imperative languages, and used the term mixed computation to mean roughly the same as partial evaluation [34, 35].

In 1984, Jones, Sestoft, and Søndergaard constructed a self-applicable partial evaluator for a simple first-order functional language [59, 60, 86]; until then neither single nor double self-application had been carried out in practice.

At the same time the interest in partial evaluation in logic programming and other areas was increasing. This was the background for the 1987 Workshop on Partial Evaluation and Mixed Computation [19, 39]. Subsequent proceedings on partial evaluation may be found in [2, 3, 4, 5, 32, 88].

9.2 Partial evaluators

Imperative languages: Early papers on partial evaluation for imperative languages include [34, 36, 37]. Bulyonkov and Ershov reported a self-applicable partial evaluator for a flow chart language [25]; so did Gomard and Jones [50]. Glück et al. created a (non-self-applicable) specializer for numeral algorithms in Fortran [11, 47]. Andersen [6, 8, 9] developed two systems for specialization of C programs; see Section 6.3.

Lisp and Scheme: The first major partial evaluator for Lisp was Redfun, reported by Beckman et al. [12]. Weise et al. constructed a fully automatic online partial evaluator for a subset of Scheme [100]. Jones et al. constructed a self-applicable partial evaluator for a first-order functional language [59, 60]; Romanenko improved it in various respects [81]. Consel constructed the self-applicable partial evaluator Schism for a Scheme subset, handling partially static structures and polyvariant binding times [28, 29, 31]. Bondorf and Danvy constructed the self-applicable partial evaluator Similix for a subset of Scheme [20, 21].

Standard ML: Danvy, Heintze, and Malmkjær developed the partial evaluator Pell-Mell [70]. Birkedal and Welinder created a generator of generating extensions [17, 18].

Refal and supercompilation: Turchin created the Refal language and developed the program transformation techniques of *driving* and *supercompilation*, which generalize partial evaluation [95, 96, 97]. A number of recent surveys on driving and supercompilation exist [48, 89, 90, 91].

Prolog partial evaluation was pioneered by Komorowski [64, 65]; subsequent work on Prolog includes [13, 44, 45, 66, 93, 98, 99]. Sahlin constructed a practical but non-self-applicable partial evaluator for full Prolog [84, 85]. Bondorf and Mogensen [76] constructed a self-applicable partial evaluator for a Prolog subset, Gurr one for the logic language Gödel [52]. Jørgensen and Leuschel created a generator of generating extensions for Prolog [62].

9.3 Related topics

McCarthy used program transformation rules in calculational proofs for recursive functional programs [72]. Boyer and Moore automated some proofs of this kind [22].

Burstall and Darlington viewed 'manual' program transformation as the application of a few types of meaning-preserving program rewritings: definition, instantiation, unfolding, folding, abstraction, and laws [26].

Partial evaluation specializes a program forwards by using knowledge about available input. Conversely, *program slicing* specializes a program backwards, using knowledge about the demand for output [79].

References

[1] S.M. Abramov and N.V. Kondratjev. A compiler based on partial evaluation. In *Problems of Applied Mathematics and Software Systems*, pages 66–69. Moscow State University, Moscow, USSR, 1982. (In Russian).

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

[3] ACM. *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*. New Haven, CT: Yale University, 1992.

[4] ACM. *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, Denmark, June 1993.

[5] ACM. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994. Technical Report 94/9, Department of Computer Science, University of Melbourne, Australia*, 1994.

[6] L.O. Andersen. Partial evaluation of C and automatic compiler generation (extended abstract). In U. Kastens and P. Pfahler, editors, *Compiler Construction, Paderborn, Germany, October 1992 (Lecture Notes in Computer Science, vol. 641)*, pages 251–257. Berlin: Springer-Verlag, 1992.

[7] L.O. Andersen. Self-applicable C program specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 54–61. New Haven, CT: Yale University, June 1992.

[8] L.O. Andersen. Binding-time analysis and the taming of C pointers. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 47–58. New York: ACM, 1993.

[9] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.

[10] P.H. Andersen. Partial evaluation applied to ray tracing. DIKU Research Report 95/2, DIKU, University of Copenhagen, Denmark, 1995.

[11] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *Partial Evaluation and Semantics-Based Program*

Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne), pages 119–132, 1994.

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

[13] K. Benkerimi and J.W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, October 1990*, pages 343–358. Cambridge, MA: MIT Press, 1990.

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

[15] A.A. Berlin. Partial evaluation applied to numerical computation. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 139–150. New York: ACM, 1990.

[16] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, University of Copenhagen, Denmark, 1993. DIKU Research Report 93/22.

[17] L. Birkedal and M. Welinder. Hand-writing program generator generators. In M.V. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94, Madrid, Spain, September, 1994. (Lecture Notes in Computer Science, Vol. 844)*, pages 198–214. Berlin: Springer-Verlag, 1994.

[18] L. Birkedal and M. Welinder. Binding-time analysis for Standard ML. *Lisp and Symbolic Computation*, 8(3):191–208, September 1995.

[19] 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.* Amsterdam: North-Holland, 1988.

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

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

[22] R.S. Boyer and J.S. Moore. Proving theorems about Lisp functions. *Journal of the ACM*, 22(1):129–144, January 1975.

[23] M. Bruynooghe, D. de Schreye, and B. Martens. A general criterion for avoiding infinite unfolding. *New Generation Computing*, 11(1):47–79, 1992.

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

[25] M.A. Bulyonkov and A.P. Ershov. How do ad-hoc compiler constructs appear in universal mixed computation processes? In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 65–81. Amsterdam: North-Holland, 1988.

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

[27] M. Codish and E. Shapiro. Compiling or-parallelism into and-parallelism. In E. Shapiro, editor, *Third International Conference on Logic Programming, London, United Kingdom (Lecture Notes in Computer Science, vol. 225)*, pages 283–297. Berlin: Springer-Verlag, 1986. Also in New Generation Computing 5 (1987) 45-61.

[28] 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. Berlin: Springer-Verlag, 1988.

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

[30] C. Consel and S.C. Khoo. Semantics-directed generation of a Prolog compiler. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP '91, Passau, Germany, August 1991 (Lecture Notes in Computer Science, vol. 528)*, pages 135–146. Berlin: Springer-Verlag, 1991.

[31] Charles Consel. *The Schism Manual, Version 1.0.* Yale University, New Haven, Connecticut, December 1990.

[32] O. Danvy, R. Glück, and P. Thiemann, editors. *Dagstuhl Seminar on Partial Evaluation, February 1996. (Lecture Notes in Computer Science, vol. xxxx).* Berlin: Springer-Verlag, 1996. (To appear).

[33] P. Emanuelson and A. Haraldsson. On compiling embedded languages in Lisp. In *1980 Lisp Conference, Stanford, California*, pages 208–215. New York: ACM, 1980.

[34] A.P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, April 1977.

[35] A.P. Ershov. Mixed computation in the class of recursive program schemata. *Acta Cybernetica*, 4(1):19–23, 1978.

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

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

[38] A.P. Ershov. On Futamura projections. *BIT (Japan)*, 12(14):4–5, 1982. (In Japanese).

[39] A.P. Ershov, D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldsson, 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).* Tokyo: Ohmsha Ltd. and Berlin: Springer-Verlag, 1988.

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

[41] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. Amsterdam: North-Holland, 1988.

[42] J. Gallagher. Transforming logic programs by specialising interpreters. In *ECAI-86. 7th European Conference on Artificial Intelligence, Brighton Centre, United Kingdom*, pages 109–122. Brighton: European Coordinating Committee for Artificial Intelligence, 1986.

[43] J. Gallagher. Tutorial on specialisation of logic programs. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. New York: ACM, 1993.

[44] J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of the Second*

Workshop on Meta-Programming in Logic, April 1990, Leuven, Belgium, pages 229–246. Department of Computer Science, KU Leuven, Belgium, 1990.

[45] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6(2,3):159–186, 1988.

[46] R. Glück and J. Jørgensen. Generating optimizing specializers. In *IEEE Computer Society International Conference on Computer Languages, Toulouse, France, 1994*, pages 183–194. IEEE Computer Society Press, 1994.

[47] R. Glück, R. Nakashige, and R. Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Dolevzal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman and Hall, 1995.

[48] R. Glück and V.F. Turchin. Application of metasystem transition to function inversion and transformation. In *International Symposium on Symbolic and Algebraic Computation, ISSAC '90, Tokyo, Japan*, pages 286–287. New York: ACM, 1990.

[49] C. Goad. Automatic construction of special purpose programs. In D.W. Loveland, editor, *6th Conference on Automated Deduction, New York, USA (Lecture Notes in Computer Science, vol. 138)*, pages 194–208. Berlin: Springer-Verlag, 1982.

[50] C. K. Gomard and N. D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, Amsterdam: North-Holland, 1989.

[51] 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.

[52] C A Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Goedel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[53] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. ACM, Berlin: Springer-Verlag, 1991.

[54] C.K. Holst. Finiteness analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 473–495. ACM, Berlin: Springer-Verlag, 1991.

[55] C.K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

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

[57] 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. Amsterdam: North-Holland, 1988.

[58] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[59] 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. Berlin: Springer-Verlag, 1985.

[60] 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.

[61] 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. New York: ACM, 1992.

[62] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. Technical Report CW 221, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 1996.

[63] S.C. Kleene. *Introduction to Metamathematics.* Princeton, NJ: D. van Nostrand, 1952.

[64] H.J. Komorowski. *A Specification of an Abstract Prolog Machine and Its Application to Partial Evaluation.* PhD thesis, Linköping University, Sweden, 1981. Linköping Studies in Science and Technology Dissertations 69.

[65] 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.

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

[67] J. Launchbury. A strongly-typed self-applicable partial evaluator. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 145–164. ACM, Berlin: Springer-Verlag, 1991.

[68] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[69] 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, MA: MIT Press, 1964.

[70] K. Malmkjær, N. Heintze, and O. Danvy. ML partial evaluation using set-based analysis. In *1994 ACM SIGPLAN Workshop on ML and Its Applications, Orlando, Florida, June 1994 (Technical Report 2265, INRIA Rocquencourt, France)*, pages 112–119, 1994.

[71] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *ICLP'95, Twelfth International Conference on Logic Programming, Tokyo, Japan, June 1995*, pages 597–611. MIT Press, 1995.

[72] J. McCarthy. A basis for a mathematical theory of computation. In P. Brafford and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. Amsterdam: North-Holland, 1964.

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

[74] T. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 116–121. New Haven, CT: Yale University, 1992.

[75] T. Mogensen. Self-applicable online partial evaluation of pure lambda calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 1995*. New York: ACM, 1995.

[76] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *LOPSTR 92. Workshops in Computing*. Berlin: Springer-Verlag, January 1993.

[77] F.G. Pagan. Converting interpreters into compilers. *Software — Practice and Experience*, 18(6):509–527, June 1988.

[78] F.G. Pagan. Comparative efficiency of general and residual parsers. *Sigplan Notices*, 25(4):59–65, April 1990.

[79] T. Reps and T. Turnidge. Program specialization via program slicing. In Danvy et al. [32]. To appear.

[80] G. Richardson. The realm of Nevryon. *Micro User*, June 1991.

[81] 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. Amsterdam: North-Holland, 1988.

[82] E. Ruf and D. Weise. On the specialization of online program specializers. *Journal of Functional Programming*, 3(3):251–281, July 1993.

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

[84] 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. Cambridge, MA: MIT Press, 1990.

[85] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, 1991. Report TRITA-TCS-9101.

[86] P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985 (Lecture Notes in Computer Science, vol. 217)*, pages 236–256. Berlin: Springer-Verlag, 1986.

[87] P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. Amsterdam: North-Holland, 1988.

[88] P. Sestoft and H. Søndergaard, editors. *Special Issue on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'94). (Lisp and Symbolic Computation, vol. 8, no. 3)*, 1995.

[89] M.H. Sørensen. Turchin's supercompiler revisited. Master's thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/9.

[90] M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *International Logic Programming Symposium*, page to appear. Cambridge, MA: MIT Press, 1995.

[91] M.H. Sørensen, R. Glück, and N.D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pages 485–500. Berlin: Springer-Verlag, 1994.

[92] M. Sperber. How to have your cake and eat it, too: Self-applicable online partial evaluation. In Danvy et al. [32]. (To appear).

[93] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta programming. In H.-J. Kugler, editor, *Information Processing 86, Dublin, Ireland*, pages 415–420. Amsterdam: North-Holland, 1986.

[94] V.F. Turchin, editor. *Basic Refal and Its Implementation on Computers*. Moscow: GOSSTROI SSSR, TsNIPIASS, 1977. (In Russian).

[95] V.F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, February 1979.

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

[97] V.F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, July 1993.

[98] R. Venken. A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query-optimisation. In T. O'Shea, editor, *ECAI-84, Advances in Artificial Intelligence, Pisa, Italy*, pages 91–100. Amsterdam: North-Holland, 1984.

[99] R. Venken and B. Demoen. A partial evaluation system for Prolog: Some practical considerations. *New Generation Computing*, 6(2,3):279–290, 1988.

[100] 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. Berlin: Springer-Verlag, 1991.