Lisp and Symbolic Computation, 2, 9–50 (1989)
© 1989 Kluwer Academic Publishers—Manufactured in The Netherlands.

MIX: A SELF-APPLICABLE PARTIAL EVALUATOR FOR EXPERIMENTS IN COMPILER GENERATION

NEIL D. JONES, PETER SESTOFT and HARALD SØNDERGAARD*
DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
(neil@diku.dk, sestof+@diku.dk, harald@munnari.au)

Abstract. The program transformation principle called partial evaluation has interesting applications in compilation and compiler generation. Self-applicable partial evaluators may be used for transforming interpreters into corresponding compilers and even for the generation of compiler generators. This is useful because interpreters are significantly easier to write than compilers, but run much slower than compiled code. A major difficulty in writing compilers (and compiler generators) is the thinking in terms of distinct binding times: run time and compile time (and compiler generation time). The paper gives an introduction to partial evaluation and describes a fully automatic though experimental partial evaluator, called mix, able to generate stand-alone compilers as well as a compiler generator. Mix partially evaluates programs written in Mixwell, essentially a first-order subset of statically scoped pure Lisp. For compiler generation purposes it is necessary that the partial evaluator be self-applicable. Even though the potential utility of a self-applicable partial evaluator has been recognized since 1971, a 1984 version of mix appears to be the first successful implementation. The overall structure of mix and the basic ideas behind its way of working are sketched. Finally, some results of using a version of mix are reported.

Since the early 1970s it has been known that in theory, the program transformation principle called *partial evaluation* can be used for compiling and compiler generation, and even for the automatic generation of a compiler generator. A partial evaluator able to generate stand-alone compilers and compiler generators had not, however, been successfully implemented before 1984 when the first mix system was brought to work at the University of Copenhagen.

In this paper we discuss partial evaluation and its applications to compiler generation and sketch the partial evaluator we developed, called the mix system. The results we report are sufficiently remarkable to justify further research into using partial evaluation for compiler generation purposes. We also mention other applications. The description here is essentially a snapshot of the mix system and its applications as of early 1987.

A partial evaluator may be thought of as a "smart interpreter." If an ordinary interpreter is given a program and only *part* of this program's input data, it will leave

*Current address: Computer Science Department, University of Melbourne, Parkville, Victoria 3052, Australia.

the program unevaluated and report an error. A partial evaluator will attempt to evaluate the given program as far as the available input allows, yielding a new program as result.

In our terminology, partial evaluation of a *subject program* with respect to known values of some of its input parameters results in a *residual program.* Running a correct residual program on any remaining input yields the same result as running the original subject program on all of its input. Thus, a residual program is a *specialization* of the subject program to known, fixed values of some of its parameters. A *partial evaluator* is a program that performs partial evaluation given a subject program and fixed values for some of its parameters.

The relevance of partial evaluators for compilation, compiler generation, and compiler generator generation stems from the following fact. Consider an interpreter for a given programming language S. The result of specializing of this interpreter to a known source program s (written in S) *already* is a target program for s, written in the same language as the interpreter. Thus, partial evaluation of an interpreter with respect to a fixed source program amounts to compilation of the source program. From this viewpoint, then, partial evaluation and compilation are nothing but special cases of program transformation for the purpose of optimization.

Furthermore, partially evaluating a partial evaluator with respect to a fixed interpreter yields a compiler for the language implemented by the interpreter. And, even more mind-boggling, partially evaluating the partial evaluator with respect to itself yields a compiler generator, namely, a program that transforms interpreters into compilers. We return to these applications in Section 3.

It is nearly always easier to implement a new language by writing an interpreter than by writing a compiler for the language since in the latter case, one has to think of two binding times, compile time and run time. Interpretive implementations have only one binding time, but are often too inefficient for practical use. One potential significance of a good partial evaluator thus is that it allows for the automatic construction of efficient compilers from more intelligible interpretive specifications of programming languages. This is achieved by the automatic splitting of the interpreter's single binding time into two: compile time and run time. (This is called a "staging transformation" in Jørring and Scherlis [27].

The improvement is potentially very large, since it is not unusual for an interpreter to spend only a very small fraction of its time performing the operations required by the program being interpreted, the remaining time being used for various sorts of bookkeeping.

It could be argued that the restriction to language definitions in interpretive form is too limiting, since in practice one often chooses to define languages by denotational or axiomatic semantics, rather than operationally. However, denotational semantics may provide executable specifications of programming languages, as is shown by the existence of several semantics-based compiler generators that may in principle all be regarded as interpreters (we discuss this in Section 9). In such applications the potential improvement is even larger than with traditional interpreters.

The interesting question of course is basically empiric: How good are the programs that may be generated by partial evaluation techniques? Our experience (based on simple but nontrivial languages) is that mix-produced compilers turn out to be natural in structure, reasonably efficient, and able to produce efficient target programs that run up to one order of magnitude faster than the interpreted source programs. We demonstrate this by an example in Section 5 and various tables in Section 8.

The paper is organized as follows. The first three sections set up a formal framework and use it to define partial evaluation and discuss its applications to compilation and compiler generation. Other applications are barely touched upon, but a bit is said in Section 4, which lists related work, and in Section 9. Mix partially evaluates programs in a programming language called Mixwell and is itself written in this language. We introduce Mixwell in Section 5. An analysis of the problems in partially evaluating first-order functional languages is undertaken in Section 6. In Section 7 we outline the structure of mix, and the paper is concluded by an assessment in Section 8 and a discussion of directions for future research in Section 9.

Both practice and theory of partial evaluation would certainly benefit from further experimental work. The current version of mix is available from the University of Copenhagen through the authors.

1. Preliminaries

In this section a framework is set up for discussing partial evaluation and its applications. Our definition of a programming language may appear to be a bit pedantic at first sight. A precise notation is necessary, however, since more than one language may be discussed at the same time, and programs can play multiple roles: sometimes as active agents, sometimes as passive data, and sometimes even as both at once. The following definitions are inspired by both recursive function theory and Lisp. In recursive function theory, programs in the form of numerical indexes are handled as both active agents (functions) and passive objects (data) in ways resembling ours. In Lisp, programs are data structures, thus avoiding the need for the complex encodings typically used in recursive function theory. Connections between our formulations and recursive function theory will be discussed further in Section 4.

We assume there is given a fixed set D whose elements may represent programs in various languages, as well as their input and their output. The set D should be closed under formation of sequences $\langle d_1, \ldots, d_n \rangle$ of elements of D, and may be the set of all Lisp lists, for instance.

Parentheses will usually be put to use only when necessary to disambiguate expressions. We write $X \to Y$ to denote the set of all total functions from X to Y, and $X \dashrightarrow Y$ for the partial functions. A function-type expression $X \to Y \to Z$ is parenthesized as $X \to (Y \to Z)$, and a double function application f x y is parenthesized (f x) y (where f, x, and y have types f: $X \to Y \to Z$, x: X, y: Y for some X, Y, and Z).

We identify a *programming language* L with its semantic function on whole programs:

$$L: D \dashrightarrow D \dashrightarrow D.$$

The *well-formed* L-programs are those to which L assigns a meaning, i.e, L-programs = domain L.

The *input-output function* computed by $\ell \in$ L-programs is $(L\ \ell): D \dashrightarrow D$ (which is partial since ℓ may loop). Thus, $L\ \ell\ \langle d_1, \ldots, d_n \rangle$ denotes the output (if any) obtained by running the L-program ℓ on input data $\langle d_1, \ldots, d_n \rangle$. For an example, consider the following program "power" to compute x to the nth power:

```
        ┌──────────────────────────────────────────────┐
        │ f(n, x) = if n = 0 then 1                    │
power = │           else if even(n) then f(n/2, x)^2   │
        │           else x*f(n-1, x)                   │
        └──────────────────────────────────────────────┘
```

The result of running the program power is the result of applying its first function f (the goal function) to the program's input values. For example, L power $\langle 3, 2 \rangle = 8$. We take L power d to be undefined if d is not a list of length two, both of whose elements are positive integers. In Sections 2 and 3, the equality sign always means strong equality: Either both sides are undefined, or else they are defined and equal.

2. Partial evaluation

We proceed to give formal definitions of residual programs and partial evaluation.

Definition 2.1. Let ℓ be an L-program and let $d_1, d_2 \in D$. Then an L-program r is a *residual program for ℓ with respect to* d_1 iff for all $d_2 \in D$,

$$L\ \ell\ \langle d_1, d_2 \rangle = L\ r\ d_2. \qquad \square$$

Definition 2.2. A P-program p is an L-*partial evaluator* iff

$P\ p\ \langle \ell, d_1 \rangle$ is a residual L-program for ℓ with respect to d_1

for all L-programs ℓ and values $d_1 \in D$. We refer to the program ℓ as the *subject program.* □

So a partial evaluator takes a subject program and part of its input and produces a residual program; the residual program applied to any remaining input produces the same result as the subject program applied to all of its input. A consequence of Definition 2.2 is that the following characteristic equation for the partial evaluator p holds:

$$\mathsf{L}\ \ell\ \langle \mathsf{d}_1, \mathsf{d}_2\rangle = \mathsf{L}\ (\mathsf{P}\ \mathsf{p}\ \langle \ell, \mathsf{d}_1\rangle)\ \mathsf{d}_2. \tag{2.1}$$

For a simple example, let the L-program ℓ be power from Section 1, and suppose we are given that n equals 5. A trivial residual program resid_1 may easily be constructed by adding a single equation to power, resulting in the program below:

$$\mathsf{resid}_1 = \boxed{\begin{array}{ll} \mathsf{g(x)} & = \mathsf{f(5,x)} \\ \mathsf{f(n, x)} & = \textit{if}\ \mathsf{n} = 0\ \textit{then}\ 1 \\ & \quad \textit{else if}\ \mathsf{even(n)}\ \textit{then}\ \mathsf{f(n/2, x)}^2 \\ & \quad \textit{else}\ \mathsf{x} * \mathsf{f(n\text{-}1, x)} \end{array}}$$

The general possibility of partial evaluation in recursive function theory is known as the S-m-n theorem, discussed in Section 4 and traditionally proved in just this way, by adding equations.

A less trivial residual program may be obtained by symbolic evaluation of the program resid_1. This is possible since the program's control flow is completely determined by n, and it yields an equivalent program with only one equation:

$$\mathsf{resid}_2 = \boxed{\mathsf{g(x)} = \mathsf{x} * (\mathsf{x}^2)^2}$$

Partial evaluation thus can be viewed as substitution of known values for some parameters, possibly followed by equivalence preserving program transformations. It can result in residual programs which are faster (though sometimes larger) than the original. Examples may be found in Beckman et al. [2] and in Emanuelson and Haraldsson [13].

Partial evaluation followed by evaluation of the resulting program may be faster than normal evaluation because of the optimizing transformations. Actually this phenomenon occurs in the example runs described in Section 8. This should not be surprising. For example, it is often faster to compile and then run the resulting target program than to interpret.

We should stress that Definition 2.2 does not say anything about the optimizing power of a partial evaluator. The quality and efficiency of the residual programs produced by a partial evaluator wholly depends on the transformations built into the partial evaluator and on its strategies for applying the transformations.

In practice it seems difficult to obtain efficient residual programs without compromising termination properties. For instance, a residual program in a call-by-value language may terminate more often than the original subject program because of the call-by-name nature of symbolic evaluation.

When evaluated with call-by-value, the example program below will loop for all inputs (x,y):

```
f(x, y) = head(pair(x, g(x)))
g(x)    = g(x)
```

But given that x = 7, say, the program can be partially evaluated, using the "obvious" reduction of head(pair(x, g(x))), to given a program that terminates for all inputs y and returns 7:

```
f_7(y) = 7
```

We will accept this as a residual program, although it would not be a correct one according to Definition 2.1: it terminates too often. However, this is not considered a serious problem in applications.

Further, it is difficult to make a partial evaluator do powerful transformations *and* terminate, even when applied to subject programs that always terminate themselves. Hence, Definition 2.2 needs to be relaxed in practice to say that *provided* the partial evaluator terminates, the result is a residual program. By such a relaxed definition, however, a program that loops on all input is trivially a partial evaluator, which we do not want.

It would be useful to include in the definition of partial evaluation some of its desired properties to exclude "trivial" partial evaluators. Heering uses the setting of equational logic and initial algebra specifications to give a precise meaning to the vague requirement that a partial evaluator should make maximal use of the known input. He shows that in general a finite set of reduction rules is not sufficient to reduce every open term to a (minimal) normal form [23]. As a consequence it is not possible to obtain an "optimal" partial evaluator in general. This impedes a precise and complete definition of nontrivial partial evaluation that would help us in developing one. The problems we have met in this connection will be discussed further in Section 6.

3. Compilation and compiler generation

We now turn to the applications of partial evaluation to compiler generation. First we give simple formal definitions of interpreters and compilers.

3.1. Interpreters and compilers

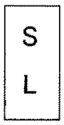
Let L and S be programming languages (S is intended to be a "source" language).

Definition 3.1. An L-program int is an S-*interpreter* iff

$$\mathsf{L\ int}\ \langle \mathsf{s, d}\rangle = \mathsf{S\ s\ d} \tag{3.1}$$

for all S-programs s and data $\mathsf{d} \in \mathsf{D}$. □

By this definition, an interpreter takes as input both the program to be interpreted and its input data. We will call int and L-*self-interpreter* iff S = L (sometimes the term "metacircular interpreter" is used). The set of interpreters for the language S (written in L) is denoted by

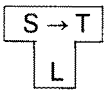


Now let T be a programming language (intended to be a target language).

Definition 3.2. An L-program c is an S-to-T *compiler* iff

1. $\mathsf{L\ c\ s} \in$ T-programs for all S-programs s, and
2. $\mathsf{T(L\ c\ s)\ d = S\ s\ d}$ for all S-programs s and data $\mathsf{d} \in \mathsf{D}$. □

The result $\mathsf{t = L\ c\ s}$ of running a compiler thus is a (target) T-program t with the same input–output behavior as the (source) program s. The set of S-to-T compilers written in L is denoted by



3.2. Compilation by partial evaluation of an interpreter

Let the P-program p be an L-partial evaluator. If an S-interpreter int is partially evaluated with respect to a given S-program s, the result will be an L-program with the same input–output behavior as s, since

$$\begin{aligned} \mathsf{S\ s\ d} &= \mathsf{L\ int}\ \langle \mathsf{s, d}\rangle && \text{by (3.1)} \\ &= \mathsf{L\ (P\ p\ \langle int, s\rangle)\ d} && \text{by (2.1)} \end{aligned}$$

Note that the last line describes the application of a certain L-program (namely the program $\mathsf{P\ p}\ \langle \mathsf{int, s}\rangle$) to the input d. The result of this is the same as the result of applying the S-program s to d, and therefore we may reasonably call the resulting program target

$$\text{target} = \text{P p} \langle \text{int, s} \rangle$$

since it is an L-program with the same input–output behavior as S-program s. In other words we have compiled the source S-program s into an L-program target by partially evaluating the S-interpreter with respect to the source program s. For concrete examples, see Figure 2 (source program s), Figure 3 (interpreter int), and Figure 4 (target program target) below.

3.3. Compiler generation

Definition 3.3. An L-program mix is an L-*autoprojector* iff it is an L-partial evaluator. We will refer to the language L as the *subject language.* □

An autoprojector is thus a partial evaluator for the language in which it is itself written. The term is from Ershov [15]: "auto" comes from the program's self-applicability, and "projector" from the fact that the residual program for f(x, y) with respect to x is a program computing a function whose graph is the projection (along the x-axis) of f's graph, in an analytical geometry sense.

In the following we will assume that a hypothetical autoprojector mix is given. By letting mix play the role of the partial evaluator p from Section 2, it holds that

$$\text{target} = \text{L mix} \langle \text{int, s} \rangle. \tag{3.2}$$

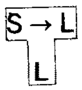
This application does not depend on mix's self-applicability, but for the following it in essential that mix is an autoprojector. A compiler from S to L may be generated by computing

$$\text{comp} = \text{L mix} \langle \text{mix, int} \rangle \tag{3.3}$$

that is, by partially evaluating the autoprojector itself with respect to the S-interpreter. To see this, observe that

$$\begin{aligned} \text{L comp s} &= \text{L (L mix} \langle \text{mix, int} \rangle\text{)s} && \text{by (3.3)} \\ &= \text{L mix} \langle \text{int, s} \rangle && \text{by (2.1)} \\ &= \text{target} && \text{by (3.2)} \end{aligned}$$

so comp is a stand-alone compiler that given s will produce a target program for s. Expressed symbolically:

comp ∈ 

Note that we now have two possibilities of compiling s by means of partial evaluation: either by running mix on ⟨int, s⟩, or by generating comp (using mix) and applying that to s. The resulting target programs will not only be equivalent; they will even be textually identical.

However, producing the target program by applying the compiler comp to s can be expected to be more efficient than by computing L mix ⟨int, s⟩. The reason is that mix is a general-purpose partial evaluator, while comp is a rather specialized version of mix, predisposed to partially evaluate a fixed interpreter int which is given varying S-programs as known input. The presumption that comp is faster is well borne out by the experimental results reported in Section 8.

3.4. Compiler generator generation*

By similar reasoning a compiler generator may be obtained:

$$\text{cogen} = \text{L mix}\,\langle \text{mix}, \text{mix} \rangle. \tag{3.4}$$

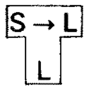
It holds that

$$\text{comp} = \text{L cogen int}.$$

To see this, observe that

$$\begin{aligned} \text{L cogen int} &= \text{L (L mix}\,\langle \text{mix}, \text{mix}\rangle)\ \text{int} && \text{by (3.4)}\\ &= \text{L mix}\,\langle \text{mix}, \text{int}\rangle && \text{by (2.1)}\\ &= \text{comp} && \text{by (3.3)} \end{aligned}$$

The function computed by the L-program cogen thus transforms an interpreter into a compiler that defines the same language:

L cogen: $\begin{matrix}\text{S}\\ \text{L}\end{matrix}$ → 

Note that this leaves us with two possible ways of producing the compiler comp: either by running mix on ⟨mix, int⟩ as in Section 3.3 or by generating cogen (by mix) and applying that to int. The resulting compilers will be textually identical in the two cases. Applying cogen is the faster of the two methods (as is illustrated by the results in Section 8).

It is interesting to compare the types of the functions computed by mix and by cogen. Let rep(A – → B) ⊆ L-programs denote the set of program representations of partial functions from A to B, and let rep(A → B) ⊆ L-programs denote the set of representations of total functions from A to B. As can be seen, the function L cogen computed by cogen is a curried version of that computed by mix:

$$\text{L mix: rep}(X \times Y - \rightarrow Z) \times X \rightarrow \text{rep}(Y - \rightarrow Z)$$

$$\text{L cogen: rep}(X \times Y - \rightarrow Z) \rightarrow \text{rep}(X \rightarrow \text{rep}(Y - \rightarrow Z)).$$

In fact, cogen is more than a compiler generator. It is a realization of a general intensional currying function, able to transform a program for a two-place function f into a program which, when given data $x = x_0$, will yield as output a program for the function $\lambda y.f(x_0, y)$. In particular, cogen transforms an interpreter into its curried form, a compiler. Also note that

$$\text{cogen} = \text{L cogen mix.}$$

In this sense cogen can be seen as a compiler generator generator generator . . . .

4. Historical notes

Theory. The concept of partial evaluation is certainly very old and has seeds from the lambda calculus and recursive function theory. To our knowledge the first explicit statement of its possibility was given when Kleene formulated and proved the S–m–n theorem [30] (see the end of this section). An early use of partial evaluation as a programming aid was suggested in Lombardi's papers on incremental computation [35,36].

Futamura saw that compiling may in principle be done by partial evaluation, and also that compilers may be generated by self-application of the partial evaluator [18]. Turchin was probably the first to realize that even a compiler generator could be built automatically by applying a partial evaluator to itself [49]. In any case these applications seem to have been independently discovered in the USSR, Japan, and Sweden in the mid 1970s and subsequently communicated in Beckman et al. [2], Ershov [14], and Turchin [50]. However, it was not until Ershov's expository paper that the ideas became widely accessible in the West [15]. Ershov coined the term "mixed computation" for what we call partial evaluation.

A consensus has been established as to how partial evaluation should be given precise definitions in imperative and functional programming. The case of logic programming has been less clear. Early attempts at a definition identified partial evaluation with a number of transformations typically employed. Recently, however, a precise "declarative" definition of partial evaluation of logic programs with negation has been suggested [34].

Practice. In the mid 1970s, projects aimed at putting partial evaluation to practical use were initiated in Sweden. A large partial evaluator for Lisp as used in practice, with imperative features and property lists, was described in Beckman et al. [2]. This work included the use of partial evaluators to translate programs in various languages, as did Haraldsson [22], Emanuelson and Haraldsson [13], and (in the United States) Turchin et al. [51]. At the same time, trends to recognize partial evaluation as an important tool appeared among dedicated builders of compiler generators [39,40].

Partial evaluation of Prolog was taken up in Komorowski [31], and Kahn developed a partial evaluator for Lisp in Prolog [28]. Partial evaluation of an imperative language was addressed in Ershov [15] and Bulyonkov [5].

The following indirect method for compiling Prolog programs was suggested by Kahn and Carlsson. A Prolog interpreter (written in Lisp) is first partially evaluated with respect to a Prolog program, yielding an equivalent Lisp program, which is then compiled into machine language using an existing Lisp compiler.

The resulting target programs are said to run faster than those produced by Warren's seminal Prolog compiler, but compilation itself is slower by two orders of magnitude [29,55].

Gallagher discussed partial evaluation of Prolog meta programs [20]. Partial evaluation can make meta programming more efficient, since the specialization in effect removes layers of interpretation.

Autoprojectors. Venken described a partial evaluator for Prolog in Prolog [52], as did Takeuchi and Furukawa, who applied theirs to the specialization of meta programs as suggested above [48]. Both were examples of *autoprojectors*, as was Safra and Shapiro's partial evaluator for Concurrent Prolog [43], one use of which was the transformation described in Codish and Shapiro [9].

To our knowledge all of these systems require considerable human assistance. None of them appear to have been successfully self-applied.

A nontrivial self-applicable partial evaluator was developed in 1984 by the authors and communicated in Jones et al. [26]. The system was called mix (following Ershov's terminology) and was a preliminary version of the fully automatic system described in the present paper. It generated good compilers by self-application, with the proviso that the user had to annotate function calls to indicate whether they were intended to be unfolded or not. A detailed description was given in Sestoft [44].

Survey. Ershov [15] and Futamura [19] are good survey papers of the area. The latter includes a bibliography. For an extensive bibliography of partial evaluation literature in English, see Sestoft and Søndergaard [46]; for one including references to papers in Russian as well, see Sestoft and Zamulin [47].

Connections with recursive function theory. The *partial recursive functions* have been studied extensively using a framework very similar to our own, but usually with function arguments, results and program encodings drawn from the natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ [41,30]. A wide variety of formalizations proposed in the 1930s as candidates to define the class of all computable partial functions have turned out to be equivalent (leading to the famous Church-Turing thesis).

In recursive function theory, one assumes given an enumeration of programs P_0, P_1, P_2, . . . , not further specified except that for each i and k there is an associated function $\varphi_i^{(k)}$: $\mathbb{N}^k \rightarrow \mathbb{N}$, namely the partial function of k arguments computed by P_i. The superscript (k) of φ_i is dropped when the number of arguments is clear from context. The Church-Turing thesis can be stated as follows: let P_i be the ith machine in a standard enumeration of all Turing machines, and let $\varphi_i^{(k)}$ be the k argument function computed by P_i. Then a partial function f: $\mathbb{N}^k - \rightarrow \mathbb{N}$ is computable iff it equals $\varphi_i^{(k)}$ for some i.

The first similarity with the framework presented in Section 1 is immediate: The given enumeration of Turing machines defines a programming language with data domain D = $\mathbb{N}$ and semantic function L: $\mathbb{N} - \rightarrow \mathbb{N} - \rightarrow \mathbb{N}$ where

$$\mathsf{L}\,\ell\,\mathsf{d} = \varphi_\ell(\mathsf{d})$$

This is extendable to multiargument functions by defining

$$\mathsf{L}\,\ell\,\langle x_1, \ldots, x_k\rangle = \varphi_\ell^{(k)}(x_1, \ldots, x_k)$$

where $\langle\_, \ldots, \_\rangle$ is one of the standard tupling functions well known in recursive function theory. The following theorems are proven in Rogers [41] for the Turing machine enumeration:

Existence of a universal machine: There exists a z such that for all x and y, $\varphi_z(x,y) = \varphi_x(y)$ if $\varphi_x(y)$ is defined, and $\varphi_z(x,y)$ is undefined if $\varphi_x(y)$ is undefined.

In our terminology the universal machine z satisfies L z $\langle z,y\rangle$ = L x y for all x, y. In other words, L-program z is an L-interpreter; a universal machine is what was called a *metacircular* or *self*-interpreter in Section 3. Another central theorem:

The S-m-n theorem: For each m,n there exists a total recursive function s_n^m of m + 1 arguments such that for all x, $y_1, \ldots, y_m, z_1, \ldots, z_n$,

$$\varphi_x(y_1, \ldots, y_m, z_1, \ldots, z_n) = \varphi_{s_n^m(x,y_1,\ldots,y_m)}(z_1, \ldots, z_n)$$

For m = n = 1, the theorem simply asserts the existence of an autoprojector, i.e., an L-partial evaluator for L. To see this, note that the recursivity of s_1^1 implies that it must have a program. Calling this "mix" and replacing φ_x by L x, the equation above is just our definition of an autoprojector:

$$\mathsf{L}x\,\langle y, z\rangle = \mathsf{L}\ (\mathsf{L}\ \mathsf{mix}\,\langle x, y\rangle)z$$

Further, any "acceptable numbering" [41, p. 41] of all recursive functions satisfies the same two theorems, so the existence of self-interpreters and a (perhaps trivial) mix program is quite natural.

The standard proof of the S–m–n theorem in essence uses the trivial construction of Section 2, which suffices for the purposes of recursive function theory. Our goals are more ambitious: to ensure that $s_n^m(x, y_1, \ldots, y_m)$ is an efficient program, and to ensure it is an efficient program even in case x is mix (self-application).

5. The language Mixwell

The choice of subject language for an autoprojector is crucial. On the one hand, the language should be simple to process. On the other hand it should be rich enough to express a nontrivial autoprojector. Any hope that a weak subject language could do must be given up because it has to be *self-interpretable.* This is due to the fact that a good autoprojector must be a generalized self-interpreter: applied to a program and all of its input, the autoprojector should do a standard evaluation, yielding a constant program as result. As a consequence of being self-interpretable, the subject language must be too complex to allow expressing its own halting function [24].

Applicative languages seem preferable to imperative ones owing to the ease with which source-to-source transformations may be performed. This is because of the property that equals can be substituted for equals without disturbing the meaning of the enclosing expression (usually referred to as referential transparency). So transformations of subterms can be done without context information.

5.1. Description of Mixwell

The subject language of mix is called Mixwell and may be thought of as essentially a subset of (pure) first-order statically scoped Lisp (or Scheme). A Mixwell program takes the form of a system of recursive equations as shown here in abstract form (examples in concrete syntax follow):

$$f_1(x_1, \ldots, x_m) = e_1$$

$$f_h(x_1, \ldots, x_p) = e_h$$

Here the f_i are function symbols, the x_j are variables (formal parameters) of the functions, and e_i is called the body expression of f_i. Expression values range over $D = \{d \mid d \text{ is a Lisp S-expression}\}$. Expressions are constructed from variables (atoms) and constants of form (*quote* d) by operators: car, cdr, cons, equal, and atom (as known from Lisp) in addition to *if* and *call.* The operator if is used in a conditional (*if* $e_0\, e_1\, e_2$), whereas *call* is used in a function call (*call* $f_j\, e_1\, e_2 \ldots e_n$). The variant *callx* is used to call external functions (e.g., gensym).

Variables have static scope. All operators are strict, except *if*, which is strict only in the first operand. In particular, *call* is strict, which implies a call-by-value semantics. Mixwell is first order: functions cannot be manipulated as data objects. The program's input is through the first function's variables.

An example Mixwell program in concrete syntax is given in Figure 1. The arguments of lookup (and hence of the program) are: a name N, a list Ns = (N_1 N_2 . . . N_n) of names, and a parallel list Vs = (V_1 V_2. . . V_n) of values. If N appears in Ns then V_i is returned, where i is the least index with N = N_i, else error is returned:

```
((lookup (N Ns Vs) = (if (equal Ns (quote nil))
                         (quote error)
                         (if (equal N (car Ns))
                            (car Vs)
                            (call lookup N (cdr Ns) (cdr Vs))))))
```

Figure 1. An example program written in Mixwell.

5.2. *Mixwell⁺*

For the sake of partial evaluation it is important that Mixwell be simple, but the above example shows that such simplicity may impair readability. We resolve this dilemma by allowing certain forms of simple syntactic extensions, translatable by machine into Mixwell. We call the extended language Mixwell$^+$. The extensions include:

- :: as an infix form of *cons*
- 'd for (*quote* d), with d $\in$ D
- (*list* e_1 e_2. . . e_N) for (e_1 :: (e_2 :: . . . (e_N :: 'nil) . . .))
- = as an infix form of *equal*, and (*null* e) for (e = '*nil*)
- a conditional
  (*if* e_1 *then* e_2 *elsf* e_3 *then* e_4. . . else e_{2N+1})
- a *case* expression
  (*case* e *of* pat_1:e_1. . . pat_N:e_N [otherwise e_{N+1}])
- *let* and *where* expressions of form
  (*let* pat_1 = e_1. . . pat_N = e_N *in* e)
  (e *where* pat_1 = e_1. . . pat_N = e_N)

Here pat_i is a pattern which is built by pairing (indicated by ".") and which contains variables that become bound to expressions selecting substructures of the value of e_i. For example, (*let* (a b) = c *in* e is equivalent to e with all free occurrences of a and b replaced by (*car* c) and (*car* (*cdr* c)), respectively. In a *case* expression two further forms of patterns are allowed: a constant pattern of form 'd, which is matched only by the S-expression d, and a pattern of form (*atom*? N) which is matched by any atom, that atom becoming bound to N as a result of matching.

5.3. An example

To give a nontrivial example program in Mixwell$^+$, we present an interpreter for another simple language M. This will also give us the opportunity to show an input/output example for a compilation done by partial evaluation of an interpreter. The language M has a syntax defined by the following grammar:

```
⟨program⟩ ::= (read  ⟨variable⟩ and evaluate ⟨expression⟩)

⟨expression⟩ ::= ⟨variable⟩
               |  (con ⟨constant⟩)
               |  (⟨operator⟩ ⟨expression⟩ ⟨expression⟩)
               |  (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
               |  (min ⟨variable⟩ such that ⟨expression⟩ = 0)

  ⟨operator⟩ ::= + | − | *

  ⟨variable⟩ ::= ⟨Lisp atom⟩
```

The intended semantics of M should be clear from the syntax. The only data type is $\mathbb{N}$, the nonnegative integers. A program reads one input value into a variable and returns the value of the body expression. For simplicity the syntactic category ⟨constant⟩ denotes lists of 1's: the nonnegative integer value m is indicated in unary as a list of m 1's. In the **if** expression, a nonzero first operand is regarded as denoting "true." The **min** expression evaluates the constituent expression for values 0, 1, . . . of the variable until the value of the expression is zero, and then returns the current value of the variable; it fails to terminate if the expression is nonzero for all values of the variable. Plus and times are the usual arithmetic operations, but note that "−" denotes *cutoff subtraction*: $x - y$ is zero if and only if $y \geqslant x$.

Figure 2 shows an example program written in M. Given a value $x \in \mathbb{N}$, the program evaluates *min* $\{y \in \mathbb{N} \mid x^2 - y^2 = 5\}$, provided the value exists.

```
(read x and evaluate
(min y such that (−(*xx)(+(*yy)(con(11111)))) = 0))
```

Figure 2. A program written in M.

Figure 3 shows an interpreter for M written in Mixwell$^+$. The kernel is the function eval which uses a traditional interpretation loop. Again, values are represented by lists. The mutually recursive functions f and g implement the "iterating" expression (**min** ⟨variable⟩ **such that** ⟨expression⟩ = **0**). To keep the example simple, very little checking is done by the interpreter: it gives meaningful results only on M

```
(
  (run (P X) = (let (read N and evaluate E) = P in (call eval E (list N) (list X))))


  (eval (E Ns Vs)
  = (case E of
      (atom? N)              : (call lookup N Ns Vs)
      ('con C)               : C
      ('+E1 E2)              : (call add (call eval E1 Ns Vs)(call eval E2 Ns Vs))
      ('-E1 E2)              : (call sub (call eval E1 Ns Vs)(call eval E2 Ns Vs))
      ('*E1 E2)              : (call mul (call eval E1 Ns Vs)(call eval E2 Ns Vs))
      ('if E0 E1 E2)         : (if (call eval E0 Ns Vs) then (call eval E1 Ns Vs))
                                                        else (call eval E2 Ns Vs))
      ('min N such that E=0): (call f E (N :: Ns)('nil :: Vs));     Bind N to 0
   otherwise                 : 'error))


  (lookup (N Ns Vs)
  = (let   (N1.Nr) = Ns
           (V1.Vr) = Vs in
    (if    (null Ns) then 'error
    elsf   (N = N1) then V1
    else   (call lookup N Nr Vr))))


  (f (E Ns Vs) = (call g (call eval E Ns Vs) E Ns Vs))          ;Evaluate E


  (g (W E Ns Vs)
  (let (V1.Vr) = Vs in
    (if    (null W) then V1                                     ;Exit if E = 0, else
    else   (call f E Ns (('1 :: V1) :: Vr)))))                  ;increment N by 1


(add(X Y) = . . .)
  (sub(X Y) = . . .)
  (mul(X Y) = . . .)
)
```

Figure 3. Interpreter for M written in Mixwell$^+$.

programs that are (syntactically) well formed, and in which every variable v used is declared by an enclosing "**min** v **such that**. . . " or "**read** v **and evaluate** . . . " expression. The M program in Figure 2 can now be translated into Mixwell by partially evaluating the interpreter in Figure 3 with P being the M program and X being unknown. As can be seen by trying out this symbolic evaluation by hand, much of the processing in the interpreter can be performed even though the input to the M program is unknown (i.e., X is unknown in the interpreter).

For example, the interpreter's main loop, including the matching of program pieces done by the case expression in function eval, can be performed completely and hence does not appear in the residual program. The same holds for the loop in the lookup function. On the other hand, the actions that depend on the unknown input to the M program cannot be performed by partial evaluation and thus must appear in the residual program. For example, the actions done in functions f and g which implement the **min** expression depend on the unknown input and cannot be performed. Hence the conditional expression from the body of the g function that appears in the residual program shown in Figure 4.

We have described the compilation from M to Mixwell by partial evaluation of the above interpreter,

target = L mix ⟨int, s⟩

where int is the interpreter in Figure 3 and s is the M source program in Figure 2. The compilation can also be done using a compiler comp produced by the mix-generated compiler generator cogen, first making the compiler,

comp = L cogen int

```
(   (run (X) = (call f (list 'nil X)))


    (f (Vs)
    = (if ('nil = (call sub (call mul (cadr Vs) (cadr Vs))
                          (call add (call mul (car Vs) (car Vs))
                                    '(1 1 1 1 1)))
          then  (car Vs)
          else  (call f (('1 :: (car Vs)) :: (cdr Vs)))))


    (add (X Y) = . . .)
    (mul (X Y) = . . .)
    (sub (X Y) = . . .) )
```

Figure 4. Target program in Mixwell$^+$.

then using it to compile

target = L comp s.

The resulting target programs will be identical. The target program shown in Figure 4 can be produced by our partial evaluator mix by either of the above methods. It is given in Mixwell$^+$ for readability.

To see how much the running time may be reduced by using the target program (Figure 4) instead of the interpreter and the source program (Figures 3 and 2), we may do a very crude time analysis.

We count one time step for each *car, cdr, cons, quote,* =, and *call*, the only exception being that the functions add, sub, and mul are counted as taking one time step for each call to any of them. By this scheme the target program executes 20 steps per iteration of its main loop, while the interpreter executes 119 steps, the ratio being 6.0, a sixfold reduction in running time.

This ratio tends to grow as the source program or the source language get larger, since relatively more interpretation time is needed for syntax analysis and environment references. Speedup factors between 30 and 200 are reported in Emanuelson and Haraldsson [13] for specialized versions of a general pattern matcher parameterized with a pattern expression to be matched.

6. Methods and problems

We now turn to the basic principles and problems involved in partially evaluating sets of recursive equations. The basic transformations used in the particular partial evaluator mix are symbolic evaluation and unfolding. These techniques are well known from the field of program transformation [7].

It should be noted that not all of the following is based on solid mathematical foundations. Some of the techniques described are heuristically based and in need of deeper analysis. Also, no completely satisfactory strategy for handling call unfolding and call specialization has been found. The automatic strategy described below works well on a large class of programs but may fail on other programs. A consequence of these problems is that the partial evaluator as implemented does not have the ideal termination properties required by Definition 2.2. It may fail to terminate even when a residual program exists, and may produce a residual program that terminates more often than the subject program.

A system-oriented mix description is given in Section 7. Note that in Sections 6–8, the language L is fixed, so L = Mixwell.

6.1. Specializing functions by symbolic evaluation

Given a subject program as a system of recursive equations, each of form

$$f(x_1, \ldots, x_n) = e$$

and given available input to this program, the residual program is naturally another system of equations, where each equation is a specialization of one of the original ones:

$$f'(y_1, \ldots, y_m) = e'.$$

Here f′ represents a *specialized* version of f, and the variables of f′ are a subset of the variables of f. For example, if it is discovered that in one call to f, the first argument always has value 5, independently of the value of the subject program's unavailable input, the partial evaluator can exploit this fact by constructing an f-variant f′ with the first variable removed, and in which e′ is a simplified version of e. A function f may have several specialized versions, each corresponding to a tuple of known values of some of its variables, or none.

The body e′ of a specialized version of a function $f(x_1, \ldots, x_n) = e$ is obtained by *symbolic evaluation* of its body expression e.

Symbolic evaluation deals with expressions (i.e., pieces of Mixwell programs) as values, and is always done in a *symbolic environment* which is a set of bindings of variables to expressions:

$$env = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}.$$

In this way, the symbolic environment in mix contains information known during function specialization about the arguments $x_1, \ldots, x_n$ of f (namely, their symbolic values). In a more sophisticated partial evaluator, symbolic environments containing other kinds of information may be used, as in Beckman et al. [2].

For a simple example of symbolic evaluation, let S⟦e⟧ represent the result of symbolically evaluating expression e in a given symbolic environment. A natural way to symbolically evaluate (*car* e) is

$$\begin{array}{lll} S[\![(\mathit{car}\ e)]\!] = \textbf{let } e' = S[\![e]\!] \textbf{ in} & & \\ \qquad \textbf{case } e' \textbf{ of} & & (6.1) \\ \qquad [\![(\mathit{quote}(x\,.\,y))]\!]: & [\![(\mathit{quote}\ x)]\!] & \\ \qquad [\![(\mathit{cons}\ e_1 e_2)]\!]: & [\![e_1]\!] & \\ \qquad \textbf{otherwise} & [\![(\mathit{car}\ e')]\!] & \end{array}$$

Note that the result is an *expression*, i.e., a piece of text. So if the result S⟦e⟧ of symbolically evaluating expression e is the expression (*cons* $e_1 e_2$) for some expressions e_1 and e_2, then S⟦(*car* e)⟧ is the expression e_1. The semantic brackets "⟦"

and "]" denote quasi-quotation as usual: Metalanguage variables (here e, e′, x, y, e_1, and e_2) may appear inside the brackets, and then stand for the expressions they are bound to, whereas object language operators stand for themselves.

If an expression e to be symbolically evaluated contains a function call, it must be decided whether the call should be *unfolded* or *suspended.* Unfolding the call means replacing it by the called function's body, with argument expressions substituted for variables. If suspended, a specialized version of the call will appear in the residual program.

Those variables of f that are present in f′ are called *dynamic* variables, and the others are called *static*. The value of a static variable is known during function specialization: It depends only on the available input. The value of a dynamic variable is considered unknown: it may depend on the unavailable input also. By the techniques used by mix, all the specialized versions of a function f have the same sequence of dynamic variables. (However, this is not necessary in principle). The generated variants of a subject program function make up a kind of tabulation of the possible values of its static variables. This technique is called *polyvariant mixed computation* by Bulyonkov [5] and is similar to function tabulation [19]. Theoretical treatments of polyvariant specialization can be found in Bulyonkov [6] and Jones [25].

6.2. *The treatment of function calls*

Some partial evaluators determine for each defined function whether all calls to it should be unfolded or suspended during partial evaluation. Other partial evaluators make this decision each time a call is encountered during function specialization. A key feature of the mix approach is to take a decision on this for each function call appearing in the text of the subject program, so that the decision may be made in advance of function specialization.

Consider a function call expression (*call* f $e_1 \ldots e_n$) to be specialized. Two obvious possibilities are either to produce a residual call (to a specialized version of f), or to unfold the call. To do the unfolding, the equation $f(x_1, \ldots, x_n) = e$ defining f is found, and (*call* f $e_1 \ldots e_n$) is replaced by the result of symbolically evaluating e in the local symbolic environment $\{x_1 \mapsto S[\![e_1]\!], \ldots, x_n \mapsto S[\![e_n]\!]\}$.

The problem. The problem of finding a good call unfolding strategy is very subtle. In this context there are at least three pitfalls to avoid.

First, a too conservative strategy leads to trivial residual programs as shown in Section 2.

Second, a too liberal strategy leads to loops during function specialization in which the same function is unfolded infinitely. In fact, only an extremely conservative strategy will avoid this danger. To see this, consider a partial evaluation with respect to a known x of

```
f(x, y) = if p(y) then x else g(0)
  g(x) = g(x + 1)
```

where p(y) in fact holds for all y, but the expression happens to be too complicated for a partial evaluator to realize this (it may express a deep number theoretic theorem). So f is total. Partial evaluation of f with respect to x, however, is likely to proceed infinitely. This will happen even in case the partial evaluator uses the very conservative rule that calls are unfolded only when all their arguments have known values. In the example, it is easy to see that g is everywhere undefined, but definedness is in general not decidable. Adopting a rule like "never unfold calls" is out of the question, because it leads to trivial partial evaluation. Therefore, one must live with the risk of nontermination or impose some termination condition, for example, stipulate some arbitrary upper limit for the number of unfoldings to be performed.

Finally, the residual programs can easily turn out to be *less* efficient than the original subject programs, owing to the call-by-name nature of symbolic evaluation. As an illustration of the last point:

```
f(n) = if n = 0 then 1 else g(f(n-1))
g(n) = n + n + 1
```

should *not* be unfolded to

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

since the first runs in linear time while the second requires exponential time. Fortunately, it is not difficult to avoid such duplicated function calls in a Lisp-like language, because the duplicates are easily recognized during symbolic evaluation. In fact, any risk of duplication can be detected even before symbolic evaluation.

Approaches to a solution. This still leaves the basic problem of when to unfold function calls. There are (at least) five possible ways. The first possibility is to unfold during function specialization only calls in which all arguments have known values, and then possibly do further unfolding in a separate stage after function specialization. This way the issues of function specialization and call unfolding are effectively separated, and the method is quite safe. Unfortunately it gives very many residual functions and has turned out to be too inefficient in our experiments. The conclusion is that also (some) calls not all of whose arguments have known values have to be unfolded during function specialization. We see four possible ways to make the decision on *which* calls to unfold:

1a. *Interactively*, during function specialization, according to advice given by the user.
1b. *By hand annotation in advance* of function specialization, individually marking all calls in the subject program as "to be unfolded" or "to be suspended."
2a. *Dynamically*, using some dynamically determined (automatic) unfolding strategy during function specialization.
2b. *By automatic static annotation*, i.e., by applying a preprocessor to mark subject program calls "to be unfolded" or "to be suspended."

Method 1a was used in the early program transformation systems, and much current research in this field concerns systematizing and automating methods that work well by hand, to reduce the complexity of what the user sees.

The first self-applicable version of mix used method 1b. However, it is difficult to see just which calls should be unfolded (and far too hard for inexperienced users). Automatic unfolding is necessary for practical use, and especially if mix-produced programs are to be partially evaluated yet further. This is because call annotation by hand then requires a user's full understanding of machine-generated programs, which is unreasonable. We therefore tried method 2a, but found it too expensive in terms of partial evaluation time.

The current version of mix uses preprocessing to add "unfold" call annotations (method 2b); then it does a straightforward symbolic evaluation as described above, blindly obeying the annotations; and finally it postprocesses the residual program so produced, to find (further) calls that can profitably be unfolded. This is all done automatically.

6.3. Some principles for call unfolding

Consider a call appearing in a recursively defined function:

$$f(x_1, \ldots, x_n) = \ldots f(e_1, \ldots, e_n) \ldots$$

If there exists an argument x_i which always decreases (according to some well-founded partial ordering), then the call may safely be unfolded, provided x_i is evaluable to a constant during function specialization. For example, the program power in Section 1 may be unfolded when n is static (has a known value), but not when n is dynamic.

This applies also to partial evaluation of interpreters. Consider the computation of

$$\text{target} = L\ \text{mix}\ \langle \text{int}, s \rangle.$$

For the great majority of programming languages, an interpreter can perform/evaluate some commands/expressions on the basis of their subcomponents, without reference to other parts of the enclosing program s (except that it uses the information carried by the environment). Thus recursive calls by the interpreter that "descend" to smaller parts of s may always safely be unfolded, but whenever the interpreter shifts its attention to a different or a larger part of s, its corresponding call may not be unfolded, owing to the risk of infinite expansion. This justifies marking individual calls rather than entire functions.

For example, in Figure 3 some of the calls implementing the **min** expressions should *not* be unfolded because of the risk of infinite expansion, but most of the others may. The problem arises because of the mixture of static actions (dependent only on the source program) and program execution actions usually found in interpreters.

One unfolding method is rather simple, but works surprisingly well in practice. A conservative strategy is used to make call annotations in a preprocessing step, and further unfoldings after function specialization are based on an analysis of the *call graph* of the first version of the residual program. We return to this analysis in Section 6.5, and a more complete description is given in Section 7.

The conservative strategy for call annotation is to mark a call as "to be suspended" unless either (1) it can be seen that all its arguments are static, or (2) a static argument is bound to a proper substructure of itself in a directly recursive call. If by this, infinite unfolding results during function specialization, then the subject program already contained a function that would be infinitely evaluated for any value of the program's dynamic parameters (though this does not imply that the subject program would run forever: the function might never be called).

Clearly some kind of program analysis is required to gather the information about which variables will be static (i.e., will have known values during function specialization). This turns out to have other uses as well, and is now described.

6.4. Preprocessing: Binding time analysis

The preprocessing, which we call *binding time analysis*, can be done by *abstract interpretation* of the subject program [11]. The program is evaluated on the two-element data domain {Static, Dynamic} to yield information about which arguments to functions will be definitely known during function specialization, and which are possibly unknown. Function variables corresponding to argument positions can thereby be classified as static or dynamic. These variable descriptions are obtained for the interpreter given in Figure 3: run(S, D), eval(S, S, D), lookup (S, S, D), f(S, S, D), g(D, S, S, D), add(D, D), sub(D, D), mul(D, D), where S = Static and D = Dynamic.

This information is used during specialization of functions and for the preprocessor's (conservative) call marking: Calls having only static arguments and calls one of whose static arguments is broken down recursively are marked for subsequent unfolding.

Further, all operators are annotated during this preprocessing, as *static* or *dynamic*, resulting in a heavily annotated version of the subject program. An operator annotated as static can be evaluated during function specialization, whereas one annotated as dynamic cannot. The intention is that the abstract interpretation yields global information about the subject program's run-time behavior, and the annotations represent this information locally. This simplifies the basic transformations used during function specialization.

6.5. Postprocessing: Call graph analysis

The strategy for marking calls during binding time analysis is rather conservative, so it is profitable to do more unfolding after the specialization phase. Hence partial evaluation has three phases: preprocessing, specialization, and final unfolding.

Due to the conservative strategy for call marking, many of the generated function body expressions will be fairly simple, often just a call to another specialized function. Such a call may be replaced by the called function's body (with appropriate substitution of argument expressions for variables), since this reduces the number of functions and calls. The call-by-name nature of such unfolding may make the residual program terminate more often than the subject program. This is regarded as of minor concern. However, for this reason the partial evaluator mix does not comply strictly with Definition 2.2 of partial evaluator.

For the final unfolding and reduction step, an analysis of the intermediate residual program must be done. This analysis works by finding a cutpoint in each elementary cycle in the program's call graph (one that does not properly contain another cycle). A cutpoint is a residual function name, and the intention is that all calls to such a function should be suspended (i.e., should not be unfolded).

Call unfolding can now be done as another symbolic evaluation: A call is suspended only if it was selected for suspension by the call graph analysis, or if unfolding would produce call duplication. By selecting a cutpoint from each elementary cycle, infinite unfolding is prevented, and hence the method is safe. More details appear in Section 7.

6.6. Special problems caused by self-application

A separate binding time analysis phase for the classification of parameters and operators is in principle unnecessary since this classification could be done dynamically (and with more precision) during the specialization phase. However, it seems to be necessary for successful and efficient self-application of the partial evaluator that the classification is determined statically, in a separate phase. Readers willing to accept this on faith may skip the rest of the section and thus escape some rather intricate and subtle argumentation.

Consider the generation of a compiler comp (from some S-interpreter int):

$$\mathsf{comp} = \mathsf{L}\ \mathsf{mix}_1\ \langle \mathsf{mix}_2, \mathsf{int}\rangle.$$

Here $\mathsf{mix}_1 = \mathsf{mix}_2 = \mathsf{mix}$—the subscripts are for reference only. Assume we determine *dynamically* (i.e., during function specialization) whether the arguments of every individual operator are static or dynamic. Now, mix_1 as well as mix_2 contain some procedure for simplifying expressions such as (*car* e), for example (6.1). Simplification depends on the residual (reduced) form S⟦e⟧ of e, which in turn depends on the form of e and the value of the subject program's known input.

The expressions occurring in mix_2 are straightforwardly reduced by mix_1, but consider mix_2 being partially evaluated on int as above. Now focus on the application of mix_2's reduction procedure for *car* on an expression (*car* e) in int. Let us assume that in int, this "*car*" is applied to int's first parameter (an S source program s).

During *compilation* one applies mix to int and a source program to get

target = L mix ⟨int, s⟩.

Thus, when the source program s is present, the *car* operator of int can be evaluated by mix. But during *compiler generation*, while generating

comp = L mix_1 ⟨mix_2, int⟩,

the source program s is not available and therefore even the *form* of the residual expression S⟦e⟧ for e in int is unknown. Therefore, the reduction procedure (in mix_2) for car cannot be executed by mix_1, and the compiler produced (i.e., the residual program for mix_2) will contain a copy of the entire reduction procedure for *car* for this single occurrence of *car* in int.

This procedure will be entirely superfluous when running the produced compiler on an S source program s, since that program will be available, and a single *car* operator could replace the reduction procedure comprising several lines of Mixwell text. In fact, the problem is even worse, because (*car*(*cdr* e)) in the interpreter int will be "reduced" to the reduction procedure for car with the entire reduction procedure for *cdr* instantiated in several places. Thus, the size of residual expressions in the compiler depends in an exponential way on the complexity of expressions in the interpreter, and this is clearly not acceptable, particularly since deeply nested *car*/*cdr* expressions are very common.

If, on the other hand, *static* operator classification is used, then it is possible to annotate int as well as mix_2. By this, a *car* operator in int working on int's static input (the S source program) will be annotated as static (as "*car* s"), and partial evaluation of mix_2 on int will produce a single *car* operator in the compiler instead of a copy of the reduction procedure. Note that the crucial point is that the annotations of int are available to mix_2, not that the annotations of mix_2 are available to mix_1. Thus, the above discussion applies only to the case when mix is itself partially evaluated; this problem really is one of self-application.

7. The algorithms used in mix

Partial evaluation using mix is most easily understood as a sequence of phases, each performing a translation, an analysis, or a transformation of the subject program (i.e., the program to be partially evaluated). In this section we first give a brief overview of the structure of mix, and then describe each of the phases. Our partial evaluation algorithm proceeds in five phases:

1. Binding time analysis	*bta*
2. Program annotation	*ann*
- -	
3. Function specialization	*fsp*
- -	
4. Call graph analysis	*cga*
5. Call unfolding and reduction	*unf*

The five phases constitute three program transformation steps, the first one and the last one consisting of an analysis phase and a synthesis phase. The purpose and input/output behavior of each is briefly described here, with more details to follow.

Suppose we want to partially evaluate an L-program ℓ (recall that L = Mixwell) with respect to known argument d_1. This yields a residual program r, satisfying $\mathsf{L}\,\mathsf{r}\,d_2 = \mathsf{L}\,\ell\ \langle d_1, d_2\rangle$ for all d_2. We will now consider the three transformations making ℓ into r.

The first transformation consists of binding time analysis and program annotation. Its output is an *annotated* version of ℓ, namely, $\ell_a = ann\langle \ell, bta\langle \ell, \mathsf{vd}\rangle\rangle$, where $bta\ \langle \ell, \mathsf{vd}\rangle$ is the information obtained by binding time analysis when the input parameters of ℓ are described by the tuple vd of {Static, Dynamic} descriptions. That is, ℓ_a is a copy of the subject program, marked with additional information:

- each function argument has been classified as static or dynamic (S or D),
- each operator (cons, car, if, etc.) has been similarly classified.
- argument lists have been permuted so all static arguments come first, and
- calls have been marked "to be unfolded" (call) or "to be suspended" (callr).

The second transformation is the function specialization phase, which is the heart of partial evaluation. It produces an intermediate residual program, r′, for ℓ_a, given the annotated program and the input available for partial evaluation.

The third transformation comprises call graph analysis together with call unfolding and reduction. To produce a (better) final residual program r, more function calls are unfolded and redundant code is reduced in the intermediate residual program. Call graph analysis of r′ yields as output a set *cga* (r′) of function names. The idea is that avoiding unfolding of calls to these will prevent infinite expansion. The final phase applies this information to r′, yielding the final residual program

$$\mathsf{r} = unf\langle \mathsf{r}', cga\ \mathsf{r}'\rangle.$$

From this description we see that the partial evaluator mix takes three arguments, not two. The three arguments are

- the subject program ℓ to be partially evaluated,
- the input description vd which is a tuple of {Static, Dynamic) descriptions,
- the values of those input parameters described as Static.

Below we describe each of the five phases of mix, and then in Section 7.6 we discuss how this multiphase partial evaluator can be self-applied.

7.1. Binding time analysis

Input is the subject program ℓ (the program to be partially evaluated) and a description of which of its parameters will be available (known) during partial evaluation and which will not. The net result of the binding time analysis is used for annotating the subject program.

Binding time analysis is based on an abstract interpretation using the two-value domain {Static, Dynamic}. The result of this analysis describes every variable x_{ij} of every function f_i as either Static or Dynamic. Here Static means that the possible values of the variable (definitely) depend only on the input available during partial evaluation. Conversely, Dynamic means that the possible values (may) depend also on input unavailable during partial evaluation.

The binding time analysis keeps a partial description of the variables, initially describing all variables as Static, except those variables of the goal function whose values are unavailable during partial evaluation. The program is abstractly interpreted starting with the goal function, and if it is discovered that a variable v currently described as Static may take on a value dependent on a Dynamic variable, then v's description is changed to Dynamic, and the descriptions of all variables that depend on v are recomputed. Every recomputation is preceded by the change of at least one variable from Static to Dynamic, and since there are finitely many variables and they never change back, the analysis will terminate.

The abstract interpretation classifies an expression e as static if and only if:

- e is a constant (*quote* d), where d is an S-expression, or
- e is a static variable, or
- e has form (*call* f $e_1 \ldots e_n$) and $e_1, \ldots, e_n$ are all static, or
- e has form (*op* $e_1 \ldots e_n$), where $op \neq call$, and $e_1, \ldots, e_n$ are all static.

7.2. The annotation phase

Input is the subject program ℓ and the variable description just computed. Output is an ℓ-version ℓ_a in which all expressions and function calls are *annotated* for use by the function specialization phase, and argument lists are permuted so all static arguments come first.

Operator annotation. An operator in an expression other than a call is marked as static if it can be evaluated during function specialization, namely, if its performance depends only on the available input. Otherwise the operator is marked as dynamic.

Static applicability of operators is determined using the variable description found by the binding time analysis. A non-call expression (*op* $e_1 \ldots e_n$) is rewritten (*ops* $e_1 \ldots e_n$) if all of $e_1 \ldots e_n$ are static, and as (*opd* $e_1 \ldots e_n$) otherwise. The only deviation from this pattern is the conditional, which is nonstrict in its second and third argument. It is rewritten as (*ifs* $e_0 e_1 e_2$) if e_0 is static, and as (*ifd* $e_0 e_1 e_2$) if e_0 is dynamic, independently of the classification of e_1 and e_2.

Function call annotations. A function call (*call* f $e_1 \ldots e_n$) is marked as unfoldable if there is no risk of infinite expansion during function specialization, and residual otherwise. A simple (and rather conservative) scheme for recognizing this is based on the concept of an *inductive variable.*

We say that a variable of a function f is inductive in a recursive call from f to itself if in that call the variable's new value is a proper substructure of its previous value. Then a call expression (*call* f $e_1 \ldots e_n$) is unfoldable if either it is a direct recursive call to f with at least one inductive variable (the rest being unchanged) among those described as Static, or if there are no variables described as Dynamic. In all other cases, the expression is replaced by (*callr* f $e_1 \ldots e_n$), indicating that the call should not be unfolded. With annotations made this way, for "reasonable programs" infinite unfolding will not take place in the function specialization phase unless this can happen independently of the values of the dynamic data.

Furthermore, care is taken that duplication of calls cannot occur during function specialization. This may require changing more calls to *callr*. The detection of such situations is a recent enhancement to mix; for a discussion and algorithms, see Sestoft [45].

7.3. The function specialization phase

Input to this phase consists of the annotated subject program ℓ_a together with actual values for some of the subject program's parameters. Output is an intermediate residual program r′ which is a system of specialized versions of ℓ's functions.

A function that is specialized from the function $f(sv_1, \ldots, sv_m, dv_1, \ldots, dv_n) = e$ has form

$$f_{svv}\ (dv_1, \ldots, dv_n) = e'$$

where f_{svv} is a new function name composed from the name f of the function in ℓ and a sequence svv of values of f's static variables $sv_1, \ldots, sv_m$. The variables of f_{svv} are f's dynamic variables $dv_1, \ldots, dv_n$, and $e' = S[\![e]\!]$ is the result of symbolically evaluating the right side e of the original function definition $f(\ldots) = e$ using the known values svv for f's static variables $sv_1, \ldots, sv_m$.

The classification of all variables as static or dynamic makes it easier to build residual programs, and the basic transformation rules become very simple. For example, the argument of *cars* will always be a known value, and so *cars* can be evaluated during partial evaluation, whereas a *card* expression will be left as residual. Therefore, the **case** expression (6.1) for $S[\![(\mathit{car}\ e)]\!]$ is replaced by two simpler rules:

$$S[\![(\mathit{cars}\ e)]\!] = [\![(\mathit{quote}\ x)]\!] \text{ where } (x\,.\,y) = S[\![e]\!], \text{ and}$$

$$S[\![(\mathit{card}\ e)]\!] = [\![(\mathit{car}\ e')]\!] \text{ where } e' = S[\![e]\!].$$

The set of function specializations is computed by maintaining a set Pending to record all the function specializations still to be computed. A pair (f, svv) being in Pending means that a version $f_{svv}(dv_1, \ldots, dv_n) = e'$ of function f specialized to the values svv of the static variables of f is needed. Initially, Pending contains one pair consisting of the goal function and the argument values that are available for partial evaluation. When a nonunfoldable call $(\mathit{callr}\ g\ se_1 \ldots se_m\, de_1 \ldots de_n)$ is symbolically evaluated, it is recorded that a new specialized function g_{svv} is needed by adding the pair (g, svv) to Pending, where $svv = (S[\![se_1]\!], \ldots, S[\![se_m]\!])$ is a list of values of g's static variables. In reality, the composite function names of form f_{svv} that may be very large and cumbersome to read are (consistently) replaced by new shorter function names.

Since Pending may grow and shrink indefinitely, the function specialization phase may loop in an attempt to produce infinitely many different specialized functions. This happens only on programs that compose static data under the control of dynamic data. Work is under way to reduce the frequency of such undesirable behavior.

7.4. Call graph analysis

Input to this phase is the intermediate residual program r′. Output is a list of function names from r′ that are cutpoints of recursive call chains. This is for use by the subsequent call unfolding and reduction phase.

The *call graph* of a program is a directed multigraph that has the program's functions as nodes and has an edge from node f to node g for each call to g in the body of f. A *recursive call chain* in a program is a cycle in the call graph.

The *cutpoints* of recursive call chains are found by a depth-first traversal of the call graph of the intermediate residual program r′. A *visit* to a function f entails marking that function "visited", and then examining all function calls in its body. Consider a call from f to a function g. If g is already on the path traversed from the goal function to f (inclusive), then g is taken to be cutpoint of a recursive call chain. If g is not on that path and has not been visited previously, then g is visited. When there are no more functions that can be visited from f, the algorithm backs up to the function from which f was visited (or terminates if f is the goal function). During this, every function is visited at most once, and hence the process terminates.

7.5. The call unfolding and reduction phase

Input to this phase consists of the intermediate residual program r′ (resulting from the function specialization phase) and the list of cutpoints of recursive call chains. Output is the final residual program, r, obtained from the intermediate residual program r′ by unfolding function calls and reducing the resulting expressions by symbolic evaluation.

A function call (*call* f $e_1 \ldots e_n$) in r′ is unfolded if f is not head of a recursive call chain and if unfolding will not lead to duplication of a function call (or of a voluminous expression) when the expressions $e_1, \ldots, e_n$ are substituted for the variables in f's body. Since infinite unfolding would involve every function in some recursive call chain, and hence a cutpoint, it cannot take place. Similarly, no call duplication can be introduced by the call unfolding phase.

7.6. Multiphase partial evaluation

We now relate this more complex picture of partial evaluation (in multiple phases) to the simpler descriptions given in Sections 2 and 3. In particular, we will show the multiphase analogs of

$$\text{target} = L\ \text{mix}\langle\text{int}, \text{s}\rangle, \tag{3.2}$$

$$\text{comp} = L\ \text{mix}\ \langle\text{mix}, \text{int}\rangle, \text{ and} \tag{3.3}$$

$$\text{cogen} = L\ \text{mix}\ \langle\text{mix}, \text{mix}\rangle. \tag{3.4}$$

Simple partial evaluation, including compilation. *Bta*, *ann*, and so forth are functions, so if we let

$$\mathit{pre}\ \langle \ell, \text{vd}\rangle = \mathit{ann}\ \langle \ell, \mathit{bta}\ \langle \ell, \text{vd}\rangle\rangle$$

and

$$\mathit{post}\ \text{r} = \mathit{unf}\langle \text{r}, \mathit{cga}\ \text{r}\rangle,$$

then we can write

$$\text{resid} = \textbf{let}\ \ell_a = \mathit{pre}\ \langle \ell, \text{vd}\rangle\ \textbf{in}$$
$$\textbf{let}\ \text{r}' = \mathit{fsp}\ \langle \ell_a, \text{d}_1\rangle\ \textbf{in}\ \mathit{post}\ \text{r}'.$$

Here vd ∈ {Static, Dynamic}* is a description of the parameters of ℓ, and the metanotation **let** . . . **in** . . . is used to indicate the partitioning into phases. It would certainly be possible to program mix so that mix yields resid in exactly this way, thus satisfying the equations of Sections 2 and 3 quite literally, by letting

$$\mathsf{L}\ \mathsf{mix}\ \langle \ell, \mathsf{vd}, \mathsf{d}_1 \rangle = \textbf{let}\ \ell_a = \mathit{pre}\ \langle \ell, \mathsf{vd} \rangle\ \textbf{in}$$
$$\textbf{let}\ \mathsf{r}' = \mathit{fsp}\ \langle \ell_a, \mathsf{d}_1 \rangle\ \textbf{in}\ \mathit{post}\ \mathsf{r}'.$$

It sometimes happens (especially in compiling, where ℓ is an interpreter and d_1 a source program) that the same program ℓ is to be partially evaluated for many different values of d_1, so it is usually profitable to compute $\ell_a = \mathit{pre}\ \langle \ell, \mathsf{vd} \rangle = \mathit{ann}\ \langle \ell, \mathit{bta}\ \langle \ell, \mathsf{vd} \rangle \rangle$ as a separate first step. Compilation, using mix on an interpreter int and source program s, is an instance of this process. That is, int_a should be precomputed, describing the first parameters as static and the second as dynamic:

$$\mathsf{target} = \textbf{let}\ \mathsf{int}_a = \mathit{pre}\ \langle \mathsf{int}, \langle \mathsf{Static}, \mathsf{Dynamic} \rangle \rangle\ \textbf{in} \tag{7.1}$$
$$\textbf{let}\ \mathsf{r}' = \mathit{fsp}\ \langle \mathsf{int}_a, \mathsf{s} \rangle\ \textbf{in}\ \mathit{post}\ \mathsf{r}'.$$

Compiler generation by partial evaluation. At compiler generation time, int is known, but s is not. As before, it is desirable to precompute $\mathsf{int}_a = \mathit{pre}\ \langle \mathsf{int}, \langle \mathsf{Static}, \mathsf{Dynamic} \rangle \rangle$, so we assume this has been done as the first step of compiler generation. The remaining computation

$$\textbf{let}\ \mathsf{r}' = \mathit{fsp}\ \langle \mathsf{int}_a, \mathsf{s} \rangle\ \textbf{in}\ \mathit{post}\ \mathsf{r}'$$

depends on both int_a and s, but only int_a is available. For efficiency we use partial evaluation to exploit the target program's dependency on int (since s changes more frequently than int). Suppose fsp is an L-program computing function *fsp*, so we have L fsp = *fsp*. Now construct

$$\mathsf{comp} = \textbf{let}\ \mathsf{int}_a = \mathit{pre}\ \langle \mathsf{int}, \langle \mathsf{Static}, \mathsf{Dynamic} \rangle \rangle\ \textbf{in}$$
$$\mathsf{L}\ \mathsf{mix}\ \langle \mathsf{fsp}, \langle \mathsf{Static}, \mathsf{Dynamic} \rangle, \mathsf{int}_a \rangle$$
$$= \textbf{let}\ \mathsf{int}_a = \mathit{pre}\ \langle \mathsf{int}, \langle \mathsf{Static}, \mathsf{Dynamic} \rangle \rangle\ \textbf{in}$$
$$\textbf{let}\ \mathsf{fsp}_a = \mathit{pre}\ \langle \mathsf{fsp}, \langle \mathsf{Static}, \mathsf{Dynamic} \rangle \rangle\ \textbf{in}$$
$$\textbf{let}\ \mathsf{r}' = \mathit{fsp}\ \langle \mathsf{fsp}_a, \mathsf{int}_a \rangle\ \textbf{in}\ \mathit{post}\ \mathsf{r}' \tag{7.2}$$

It is not hard to see (by (2.1)) that comp really is the first pass of a compiler, and

$$\text{target} = \textbf{let } r' = L \text{ comp s } \textbf{in } \textit{post } r'. \tag{7.3}$$

Finally, note that fsp_a is independent of int. In summary, we have:

- $\text{fsp}_a = \textit{pre}\,\langle \text{fsp}, \langle \text{Static, Dynamic}\rangle\rangle$ is independent of int and so may be computed prior to compiler generation (i.e., at compiler generator generation time).
- At compiler generation time we compute

$$\begin{aligned}\text{comp} = {} & \textbf{let } \text{int}_a = \textit{pre}\,\langle \text{int}, \langle \text{Static, Dynamic}\rangle\rangle \textbf{ in}\\ & \textbf{let } \text{fsp}_a = \textit{pre}\,\langle \text{fsp}, \langle \text{Static, Dynamic}\rangle\rangle \textbf{ in}\\ & \textbf{let } r' = \textit{fsp}\,\langle \text{fsp}_a, \text{int}_a\rangle \textbf{ in } \textit{post } r'.\end{aligned}$$

Generation of the compiler generator. Section 3.4 described producing a compiler generator cogen from mix by partially evaluating mix with respect to itself. By steps exactly parallel to the preceding, we obtain

$$\begin{aligned}\text{cogen} = {} & \textbf{let } \text{fsp}_a = \textit{pre}\,\langle \text{fsp}, \langle \text{Static, Dynamic}\rangle\rangle \textbf{ in}\\ & \textbf{let } r' = \textit{fsp}\,\langle \text{fsp}_a, \text{fsp}_a\rangle \textbf{ in } \textit{post } r',\end{aligned} \tag{7.4}$$

and this is in fact the way the first version of cogen was obtained. Given cogen, compiler generation may be done more efficiently than above:

$$\begin{aligned}\text{comp} = {} & \textbf{let } \text{int}_a = \textit{pre}\,\langle \text{int}, \langle \text{Static, Dynamic}\rangle\rangle \textbf{ in}\\ & \textbf{let } r' = L \text{ cogen int}_a \textbf{ in } \textit{post } r',\end{aligned} \tag{7.5}$$

and the compiler generator itself may be regenerated:

$$\begin{aligned}\text{cogen} = {} & \textbf{let } \text{fsp}_a = \textit{pre}\,\langle \text{fsp}, \langle \text{Static, Dynamic}\rangle\rangle \textbf{ in}\\ & \textbf{let } r' = L \text{ cogen fsp}_a \textbf{ in } \textit{post } r'.\end{aligned} \tag{7.6}$$

8. Assessment of the partial evaluator mix

In this section we evaluate the structure and performance of mix, and we mention some of the tasks mix has been applied to.

8.1. Ideas behind mix

The aim has been to construct an *autoprojector*, well suited for the special purpose of compiler generation, rather than a general-purpose partial evaluator. This makes the task easier in some respects, but, in general, the development of a good autoprojector is harder than that of a partial evaluator. The reason is that owing to the self-application an attempt to increase the *quality* (by including more powerful transformations) very often implies overwhelming penalties as regards *efficiency*.

The structure of mix has intentionally been kept as simple as possible, partly for this reason and partly to allow for experimentation with different binding time analyses, basic transformations, call unfolding strategies etc., and their combination.

Program transformation is concerned with deriving equivalent programs that behave better according to some performance criteria. These criteria are implicit in the transformation rules used; so the basic transformations together with the call unfolding strategy determine the strength of the partial evaluator. One may think of the transformations performed by mix as split into two categories. One consists of simple local *reductions*, while the other is concerned with function transformation and includes *unfolding* of calls and *specialization* of functions. Although unfolding and specialization constitute a limited class of transformations, they may imply considerable changes in program topology.

The use of binding time analysis appears to be novel in comparison to other approaches to program transformation. It serves three purposes: to classify function variables, thereby determining the list of residual variables for each function; to annotate all operators as "static" or "dynamic"; and to gather information used to attach unfold/suspend annotations to function calls. As a result of binding time analysis, we have been able to reduce the transformation rules used to an extremely simple subset. If binding time analysis is not applied, the generated compilers in our experiments have turned out to be typically two orders of magnitude larger, and much less efficient.

The call unfolding strategy seems appropriate, and when given suitably written subject programs, mix gives good results. The target programs and compilers produced are reasonably small and efficient. While they sometimes contain inelegant code, they contain little unnecessary code, as is witnessed by the fact that compilation speed is of the right order of magnitude, about 100 lines/second on a Vax 785 for a toy language. On the whole they look like traditional recursive descent compilers, except that more optimization is done while generating code than usual in compilers. Since the compiler is a specialized version of mix, it inherits the transformational capabilities built into the partial evaluator.

8.2. Performance of mix

To illustrate the performance of mix in compilation and compiler generation we give some tables of program size and run times. In particular, we give the total run times for each of the runs (7.1) through (7.6) discussed in section 7.6, and show how the run time is composed of preprocessing time, function specialization time, and post-processing time.

The interpreter int used in the runs below interprets a tiny imperative language (called MP) with assignment, a conditional, a while-loop, and with S-expressions as the only data type. The MP source program s used computes x to the yth power by enumerating all different tuples of length y with elements chosen from a set of cardinality x. The runs involving s and target in Figure 6 compute $5^5 = 3125$.

The size of a Mixwell program is given by two figures: the number of functions in the program, and the length in lines when translated into Lisp and "prettyprinted" (Figure 5). The programs generated by mix are seen to have a very manageable size, considering that target results from "combining" int and the source program s, that

Program	No. of functions	No. of lines	Ratio (lines)
s	–	approx. 30	1.2
target	6	36	
int	9	176	1.7
comp	24	303	
fsp	27	533	2.0
cogen	49	1062	

Figure 5. Size of programs.

Run	Run time (cpu secs.): processing + g.c. = total	Speed -up	Run No.	Plus run time for *pre* and *post*	Total	Speed -up
output = L in⟨s,data⟩	19.62 + 2.20 = 21.82	8.1			21.82	8.1
= L target data	0.56 + 2.14 = 2.70				2.70	
target = L fsp⟨int_a,s⟩	0.66 + 0.00 = 0.66	1.9	(7.1)	*pre*(int): 0.50	1.42	2.4
				post(target): 0.26		
= L comp s	0.34 + 0.00 = 0.34		(7.3)	*post*(target): 0.26	0.60	
comp = L fsp⟨fsp_a,int_a⟩	7.56 + 3.00 = 10.56	2.3	(7.2)	*pre*(int): 0.50	15.68	2.2
				pre(fsp): 2.60		
				post(comp): 2.02		
= L cogen int_a	3.18 + 1.42 = 4.60		(7.5)	*pre*(int): 0.50	7.12	
				post(comp): 2.02		
cogen = L fsp⟨fsp_a,fsp_a⟩	37.32 + 21.72 = 59.04	1.6	(7.4)	*pre*(fsp): 2.60	72.34	1.5
				post(cogen) 10.70		
= L cogen fsp_a	19.84 + 16.46 = 36.30		(7.6)	*pre*(fsp): 2.60	49.60	
				post(cogen): 10.70		

Figure 6. Run times

comp results from combining fsp and int, and that cogen results from combining two copies of fsp.

The run time results (Figure 6) were obtained with the Franz Lisp system running under Unix on a Vax 785. The Mixwell programs were (straightforwardly) translated into applicative Lisp programs and compiled to have fast (direct) function calls. Run times are given in form processing time + garbage collection time = total run time (in cpu seconds). The left side of the table shows the bare run time of the function specialization phase fsp and of the residual programs (comp and cogen) derived from it, whereas the right side gives the additional time spent on pre- and postprocessing and the total run times for the runs (7.1) through (7.6) of Section 7.6. The corresponding speed-up ratios are also given, and are seen to be all greater than 1.

Recall that the runs being compared pairwise produce *identical* results. For example, the target program generated by compilation (run (7.3)) is not only equivalent to but in fact identical to the target program generated by partial evaluation (run (7.1)). Thus, the two target programs are known to be of exactly the same efficiency and quality. The only difference is in the time it takes to generate them.

The run-time results in Figure 6 show that

- The overhead of interpretation is removed by compiling the source program s into a target program target. The speed-up is more than 8 times, which is quite satisfactory.
- Compilation by a mix-generated stand-alone compiler (7.3) is twice as fast as compilation by partial evaluation (7.1).
- Generating a compiler using the mix-generated compiler generator (7.5) is faster than generating a compiler by partially evaluating the partial evaluator with respect to the interpreter (7.2).
- Similarly, regenerating the compiler generator cogen by using the compiler generator (7.6) is faster than generating it using mix alone (7.4).

Also note that compilation by partial evaluation followed by a run of the target program (4.12 seconds in all) is faster than interpretation of the source program (21.28 seconds).

Even generation of a compiler followed by compilation and a run of the target program (10.42 seconds in all) is faster than interpretation of the source program.

The results, and in particular the run time results, justify our approach: compiling by means of a mix-generated compiler *is* faster than compiling using a general partial evaluator, as it is done by Kahn and Carlsson [29] or Takeuchi and Furukawa [48].

8.3. Applications of mix

Mix has been used on a variety of problems, all of an experimental nature but some more applied than others. Mostly it has been used to generate compilers and target programs for various languages (imperative, functional and pattern matching).

One larger application has been context-free parsing [12]. A general-purpose context-free parser resembling Earley's was partially evaluated with respect to a fixed grammar G, automatically yielding a much more efficient parser, specialized to the syntax defined by G. Further, application of cogen to the general parser yielded a parser generator:

$$\begin{aligned}\text{specificparser} &= \text{L mix } \langle \text{generalparser, G} \rangle \\ \text{parsergenerator} &= \text{L mix } \langle \text{mix, generalparser} \rangle \\ &= \text{L cogen generalparser}\end{aligned}$$

Another application has been improvement of the important but computation-intensive ray-tracing technique of computer graphics [37]. Here the ray tracer was partially evaluated with respect to a given scene. For this purpose, Mogensen has written a rather larger version of mix than the one described here, using C as implementation language instead of Lisp. The subject language is still functional and allows computation with floating point numbers. Mogensen's version of mix is also self-applicable, but does not automatically determine call unfolding. Significant improvements in computation time have been reported.

9. Perspectives and directions for future research

We conclude by putting the present work a little into perspective. We review the programming language we have used. Also, other applications are mentioned, and we discuss the practicability of some of these.

9.1. Subject language

As regards the choice of language which mix accepts and in which it is written, we think that the following characteristics of Mixwell have contributed much to the practicability of the project:

- Programs can accept programs as input data and produce them as output.
- Mixwell's simple semantics makes it easy to perform symbolic evaluation and to design a good binding time analysis. In particular, good unfolding properties seem essential.
- The recursion natural to the partial evaluation process is easy to program.
- The referential transparency of the language facilitates specialization of an arbitrary program part without disturbing other parts.

It would be very desirable to have a self-applicable partial evaluator for an imperative language, because target programs would then come out in a language that we know how to implement efficiently. It seems, however, more difficult to build an autoprojector for an imperative langauge, and the problem is still open as far as we know. One difficulty is recognizing "descents" by an imperative program into a smaller part of a structured static argument. Also, the referential opacity of such a language necessitates more sophisticated symbolic environments for use during function specialization and more sophisticated partial evaluation techniques.

A self-applicable partial evaluator for a higher-order functional language, or one for a language that includes function invocation by pattern matching would also be very desirable, owing to the power and conciseness of such languages. It would probably be harder to write than one based on Mixwell, because of more complex control flow and data descriptions needed for these.

Logic programming languages also seem to have all of the above mentioned useful characteristics, so a nontrivial self-applicable partial evaluator for Prolog should be possible. Performing constant propagation in a Prolog program is not hard, but unfolding problems become more difficult than in Mixwell, owing to Prolog's more complicated parameter concept and control flow. In particular, an automatic binding time analysis for Prolog programs could be expected to be somewhat harder.

9.2. *Meta programming without loss of efficiency*

As is well known, many programming language definitions (though not all) are effectively computable. Thus, for many well-defined classes of language definitions, one may in principle write (in some language L) a metainterpreter mint, such that given a language definition def that defines a language S and a program s in S,

$$\mathsf{L}\ \mathsf{mint}\ \langle \mathsf{def}, \mathsf{s}, \mathsf{d}\rangle = \mathsf{S}\ \mathsf{s}\ \mathsf{d}.$$

This was first established for denotational semantics in Mosses [39], with def being a lambda term, and the approach has been developed further in, e.g., Paulson [40], Christiansen and Jones [8], Vickers [53], Watt [56], and Lee and Pleban [33]. There have, however, been substantial efficiency problems with such approaches, some of which have been overcome by more or less formalized *binding time splits*, for example, by compiling def into a lower-level and more directly executable language.

Efficiency analysis. The reason why such a metainterpreter is inefficient is not hard to see. It spends most of its computational efforts scanning and decomposing def, to see which of the definition's rules to apply to execute the operations given in textual form in s. And aside from the operations that scan and decompose def, most of the remaining operations will, as in any interpreter, scan and decompose s. Only a vanishingly small fraction of mint's computational time is spent performing the computational operations actually specified by s.

Consequently, the primary efficiency barrier to overcome in implementing a language by this technique is to remove the operations needed to analyze the language definition and the program being executed, since these are uninteresting with respect to the computational task to be realized by s. In other words, a binding time split has to be done, to move these irrelevant operations to an earlier stage in the process of transforming s into executable form, so the computation that s specifies can be carried out with minimal overhead. Analogies from traditional optimizing compilers include *code motion* and *constant propagation* [1]. Notions similar to the binding time split have appeared in the area of program transformation. A generalized view leads to the notion of *staging transformations* [27], and a similar theme is developed by Wand [54].

As already mentioned, the reason for the unacceptable inefficiency of metainterpreters is the time spent in the (meta) interpretation loop. Assume that we have a fixed language definition def and program s to be implemented. Then the running time of the metainterpreter L mint ⟨def, s, d⟩ can be expected to be a *near-linear* function of the number of fundamental operations specified by the program s on given input data d (although with a very large constant multiplier).

We would expect that also the running time of a target program for s is linearly dependent on the number of operations specified by s, only with a much smaller multiplier. Our chief goal is thus to reduce a linear factor, a goal much less ambitious and, we hope, requiring considerably less sophisticated methods than those needed for program transformation in general.

Hierarchies of metalanguages. It is becoming increasingly popular to solve a wide-spectrum problem not by writing a collection of special-purpose programs, but rather by devising a *problem-oriented language* in which the user can interactively express a wide variety of computational requests. The current broad interest in developing expert systems exemplifies this way of solving problems.

A problem-oriented language needs a processor, and these processors usually work interpretively, alternating between reading and deciphering the user's requests, consulting databases, and doing problem-related computing. For some sophisticated problem-oriented languages, the system spends a considerable amount of time interpreting rather than computing or searching, and here automatic optimization of system programs could yield substantial benefits.

Further, expert and other programming systems are being constructed more and more with the use of a hierarchy of metalanguages, each used to control the sequence and choice of operations at the next lower level [43]. In this context the efficiency problem becomes more serious, and the benefits of automatic program optimization are correspondingly greater, since widespread use of meta programming can easily lead to multiple layers of interpretation, each multiplying the total computation time by an essential factor. On the other hand, program specialization can (and has been shown to) eliminate an entire level of interpretation, so that meta programming may be used without order-of-magnitude loss of efficiency [48].

9.3. *Partial evaluation and program transformation*

It is interesting to compare the state of the art in partial evaluation with that of the field of program transformation in general. It is commonly agreed that completely automated program transformation has not been achieved on a significant scale. Why then do we consider it reasonable to attempt to transform a semantic definition of an entire programming language into a prototype compiler? Are we guilty of wishful thinking, or are there some essential differences between language implementation and transformation of more general programs?

Program transformation is concerned with rather radical changes to a program's structure, so the final program may have properties very different from those of the original one. A common goal, for instance, is to change a program's running time as a function of input size, often from exponential to polynomial or from polynomial to linear.

We have argued that partial evaluation can achieve order-of-magnitude *linear* speedups (e.g., of target programs over interpreters) but it seems unlikely that partial evaluation can yield nonlinear speedups in general. One reason is that partial evaluation uses only a single transformation technique, essentially a generalization of well-known compiler optimizations.

So the goals of partial evaluation are in a sense more modest and, we think, achievable by simpler methods than those of program transformation in general.

Generation of automatic program transformers. Partial evaluation may be used to obtain program transformers in a simple way. Assume we are given a *self-interpreter* sint. Now

$$\text{tra} = \text{L mix} \langle \text{mix, sint} \rangle$$
$$= \text{L cogen sint}$$

is a source-to-source compiler or, in other words, a (machine-generated) *program transformer*. The output of tra will be a program that is functionally equivalent to that given to tra, but may differ in structure, size, efficiency, and other properties. The relation between the input and output program is determined rather implicitly, by the way sint is written and by the transformations that are built into mix.

As to potential applications of this idea, sint could be modified to accept an extended language or to do additional run-time actions, thus achieving some of the goals of meta programming. That is, tra would transform (compile) a program in the extended language into an L program, while removing those actions of the interpreter sint that can be done already at "compile time."

9.4. Current activities in partial evaluation

The work reported here is currently being extended in numerous ways: to use restricted term rewriting systems as subject language [4]; to allow for data structures that may be partially static and partially dynamic [38]; and to ensure better termination properties of partial evaluation by improved binding time analyses [25].

Work closely related to that reported here is being done by several researchers. Romanenko has implemented a self-applicable partial evaluator similar to the one reported here, improving it in various ways, except that hand-made call annotations are still needed [42]. Similarly, Consel reports a self-applicable partial evaluator for an extendible first-order subset of Scheme without side effects. It incorporates so-called filters, which are hand-made annotations that allow the user to guide partial evaluation with high precision [10].

Recently, successful self-application of Prolog partial evaluators has been reported [16,17]. As with previous versions of mix, annotations are called for to guide the partial evaluator, so *automatic* compiler generation has not quite been achieved using Prolog partial evaluators, but that gap may be closing. Recent activities in partial evaluation of logic programs (besides those just mentioned) are represented by the ten papers in a special issue of *New Generation Computing* **6**, (2, 3), June 1988.

The selection of papers in Bjørner et al. [3] represents other current activities in partial evaluation.

10. Conclusion

We have discussed partial evaluation of programs in statically scoped Lisp-like languages and described a fully automatic self-applicable partial evaluator, mix, that has been successfully applied to generate compilers for small languages, and even to generate a compiler generator. We assessed mix and gave tables of running times and space usage to illustrate its behavior.

As a basis for this, we introduced a formal framework for partial evaluation, compilation, and compiler generation which enabled the presentation of mix's applications. We also described the language Mixwell that was designed as the subject language for mix. Finally we discussed further applications and problems in the area of partial evaluation.

Acknowledgments

Many people have contributed to the mix project in many different ways. Special thanks go to Nils Andersen, Anders Bondorf, Olivier Danvy, Andrei Ershov, Carsten Gomard, Niels Carsten, Kehler Holst, Kim Høglund, Torben Mogensen, Marek Ryčko, Carolyn Talcott, Rodney Topor, and Valentin Turchin.

References

1. Aho, A. V., Sethi, R., and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, Massachusetts, 1986.
2. Beckman, L., Haraldsson, A., Oskarsson, Ö., and Sandewall, E. A partial evaluator, and its use as a

programming tool. *Artificial Intelligence* 7, 4 (1976), 319–357.
3. Bjørner, D., Ershov, A. P., and Jones, N. D. (Eds.). *Partial Evaluation and Mixed Computation*, Gl. Avernæs, Denmark, 1987. North-Holland, Amsterdam, 1988 (to appear).
4. Bondorf, A. Towards a self-applicable partial evaluator for term rewriting systems. In [3].
5. Bulyonkov, M. A. Polyvariant mixed computation for analyzer programs. *Acta Informatica* **21** (1984), 473–484.
6. Bulyonkov, M. A. A theoretical approach to polyvariant mixed computation. In [3].
7. Burstall, R. M., and Darlington, J. A transformational system for developing recursive programs. *Journal of the ACM* **24** (1977), 44–67.
8. Christiansen, H., and Jones, N. D. Control flow treatment in a simple semantics-directed compiler generator. *Proc IFIP WG 2.2: Formal Description of Programming Concepts II* (D. Bjørner, Ed.), North-Holland, Amsterdam, 1983, pp. 73–99.
9. Codish, M., and Shapiro, E. Compiling or-parallelism into and-parallelism. Proc. Third International Conference on Logic programming, London, United Kingdom, (E. Shapiro, Ed.), *Lecture Notes in Computer Science*, Vol. 225, Springer-Verlag, New York, 1986, pp. 283–297.
10. Consel, C. New insights into partial evaluation: the Schism experiment. ESOP '88, European Symp. on Programming, Nancy, France (H. Ganzinger, Ed.). *Lecture Notes in Computer Science*, Vol. 300, Springer-Verlag, New York, 1988, pp. 236–246.
11. Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Proc. Fourth ACM Symp. Principles of Programming Languages, Los Angeles, California 1977, pp. 238–252.
12. Dybkjær, H. Parsers and partial evaluation: An experiment. DIKU student report No. 85-7-15, University of Copenhagen, Denmark, 1985.
13. Emanuelson, P., and Haraldsson, A. On compiling embedded languages in Lisp. Proc. 1980 Lisp Conference, Stanford, California, 1980, pp. 208–215.
14. Ershov, A. P. On the essence of compilation. *Formal Description of Programming Concepts* (E. J. Neuhold, Ed.). North-Holland, Amsterdam, 1978, pp. 391–420.
15. Ershov, A. P. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science* **18** (1982), 41–67.
16. Fujita, H., and Furukawa, K. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing* **6** (2, 3), (June 1988) (to appear).
17. Fuller, D. A., and Abramsky, S. Mixed computation of Prolog programs. *New Generation Computing* **6** (2, 3) (June 1988) (to appear).
18. Futamura, Y. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls* **2** (5) (1971), 45–50.
19. Futamura, Y. Partial computation of programs. Proc. RIMS Symp. Software Science and Engineering (E. Goto et al., Eds.). *Lecture Notes in Computer Science*, Vol. 147, Springer-Verlag, 1983, pp. 1–35.
20. Gallagher, J. Transforming logic programs by specialising interpreters. ECAI-86, Proc. 7th European Conference on Artificial Intelligence, 1986, pp. 109–122.
21. Ganzinger, H., and Jones, N. D. (Eds.). Programs as Data Objects, Copenhagen, Denmark. *Lecture Notes in Computer Science*, Vol. 217. Springer-Verlag, New York, 1986.
22. Haraldsson, A. A partial evaluator, and its use for compiling iterative statements in Lisp. Proc. Fifth ACM Symp. Principles of Programming Languages, Tucson, Arizona, 1978, pp. 195–202.
23. Heering, J. Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science* **43** (1986), 149–167.
24. Hoare, C. A. R., and Allison, D. Incomputability. *Computing Surveys* **4** (3) (1972), 169–178.
25. Jones, N. D. Automatic program specialization: A re-examination from basic principles. In [3].
26. Jones, N. D., Sestoft, P., and Søndergaard, H. An experiment in partial evaluation: The generation of a compiler generator. Rewriting Techniques and Applications, Dijon, France (J.-P. Jouannaud, Ed.). *Lecture Notes in Computer Science*, **202**. Springer-Verlag, New York, 1985, pp. 124–140.
27. Jørring, U., and Scherlis, W. L. Compilers and staging transformations. Proc. Thirteenth ACM Symp. Principles of Programming Languages, St. Petersburg, Florida, 1986, pp. 86–96.
28. Kahn, K. M. A partial evaluator of Lisp programs written in Prolog. Proc. First Int. Logic Programming Conf., Marseille, France, 1982 (M. Van Caneghem, Ed.), pp. 19–25.

29. Kahn, K. M., and Carlsson, M. The compilation of Prolog programs without the use of a Prolog compiler. Proc. Int. Conf. Fifth Generation Computer Systems, Tokyo, Japan, 1984, pp. 348–355.
30. Kleene, S. C. *Introduction to Metamathematics*, Van Nostrand, New York, 1952.
31. Komorowski, H. J. A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation. *Linköping Studies in Science and Technology Dissertations* Vol. 69. University of Linköping, Sweden, 1981.
32. Kugler, H.-J. (Ed.), *Information Processing 86*, Proc. IFIP 86 Conf. North-Holland, Amsterdam, 1986.
33. Lee, P., and Pleban, U. A realistic compiler generator based on high-level semantics. Proc. Fourteenth ACM Symp. Principles of Programming Languages, Munich, FRG, 1987, pp. 284–295.
34. Lloyd, J. W., and Shepherdson, J. C. Partial evaluation in logic programming, Technical Report CS-87-09, Department of Computer Science, University of Bristol, England, 1987.
35. Lombardi, L. A. Incremental computation. *Advances in Computers* Vol. 8, (F. L. Alt and M. Rubinoff, Ed.), Academic, New York, 1967, pp. 247–333.
36. Lombardi, L. A., and Raphael, B. Lisp as the language for an incremental computer. In *The Programming Language Lisp: Its Operation and Application* (E. C. Berkeley and D. G. Bobrow, Eds.). MIT Press, Cambridge, Massachusetts, 1964, pp. 204–219.
37. Mogensen, T. The Application of Partial Evaluation to Ray-Tracing. Master's thesis, University of Copenhagen, Denmark, 1986.
38. Mogensen, T. Partially static structures in a self-applicable partial evaluator. In [3].
39. Mosses, P. D. SIS—Semantics Implementation System, Reference Manual and User Guide, DAIMI Report MD-30, University of Aarhus, Denmark, 1979.
40. Paulson, L. A semantics-directed compiler generator. Proc. Ninth ACM Symp. Principles of Programming Languages, Albuquerque, New Mexico, 1982, pp. 224–233.
41. Rogers, H. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.
42. Romanenko, S. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In [3].
43. Safra, S., and Shapiro, E. Meta interpreters for real. In [32] pp. 271–278.
44. Sestoft, P. The structure of a self-applicable partial evaluator. In [21] pp. 236–256.
45. Sestoft, P. Automatic call unfolding in a partial evaluator. In [3].
46. Sestoft, P., and Søndergaard, H. A bibliography on partial evaluation. *SIGPLAN Notices* **23** (2) (February 1988), 19–27.
47. Sestoft, P., and Zamulin, A. V. Annotated bibliography on partial evaluation and mixed computation. In [3].
48. Takeuchi, A., and Furukawa, K. Partial evaluation of Prolog programs and its application to meta programming. In [32] pp. 415–420.
49. TsNIPIASS. *Bazisnyi Refal i yego Realizatsiya na Vychislitelnykh Mashinakh.* TsNIPIASS, Gosstroi SSSR, Moscow, 1977.
50. Turchin, V. F. A supercompiler system based on the language Refal. *SIGPLAN Notices* **14** (2) (1979), 46–54.
51. Turchin, V. F., Nirenberg, R. M., and Turchin, D. V. Experiments with a supercompiler. Proc. 1982 ACM Symp. Lisp and Functional Programming, Pittsburgh, Pennsylvania, 1982, pp. 47–55.
52. Venken, R. A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query-optimisation. *Proc. ECAI-84, Pisa, Italy* (T. O'Shea, Ed.), North-Holland, Amsterdam, 1984, pp. 91–100.
53. Vickers, T. Quokka: A translator generator using denotational semantics. *Australian Computer Journal* **18** (1) (1986), 9–17.
54. Wand, M. From interpreter to compiler: A representational derivation. In [21] pp. 306–324.
55. Warren, D. Implementing Prolog—Compiling Predicate Logic Programs, DAI Research Report 39–40, University of Edinburgh, Scotland, 1977.
56. Watt, D. A. Executable semantic descriptions. *Software—Practice and Experience* **16** (1) (1986), 13–43.