Mix Ten Years Later

Neil D. Jones
DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
E-mail: neil@diku.dk

Abstract

The first article reporting a running nontrivial self-applicable partial evaluator appeared in1985 [18]. It described the results of one year's intense work done by Peter Sestoft, Harald Søndergaard, and myself at DIKU; work which has since led to much activity worldwide. Its significance was that it for the first time demonstrated *in practice* the feasibility of generating both compilers and a compiler generator by self-application of a partial evaluator: a possiblity which had been foreseen *in principle* by Yoshihiki Futamura in 1971, and independently by others soon after.

Partial evaluation is undergoing rapid expansion in many places in the world, with a continuing stream of new techniques, applications and even fundamental insights. This paper gives my view of some important developments that have taken place within this field over the past decade. Other researchers may well have different versions reflecting their own philosophical visions and emphases, so this paper should be taken as "the truth, the whole truth, and my version of the truth." Parts of it are adapted from [6].

1 About partial evaluation

Partial evaluation (PE for short) is *program specialization*. Its aim is to use the computer to make programs run faster, in effect mechanically trading off *program generality* in order to gain *efficiency*. In the beginning PE was just an intellectual curiosity; but has since evolved into an automatic tool for program transformation and optimization that arguably should be part of a "programming workbench".

In the past decade interest and activity in PE has grown very rapidly. Essentially similar concepts are regularly reinvented in differents places and subfields of Computer Science — a sure sign of an emerging significant body of ideas.

Practical potential. It is unfortunate that programming can be described, not unjustly, as a cottage industry: as much of a handcraft as knitting, and *with very little in the way of machinery*. Programming requires enormous amounts of human work to write, to maintain, and to adapt programs. This labor intensive practice was aptly described at the 1986 IFIP Congress as "high technology basket weaving" by Bill Wulf, a world leader in software development.

Today's approaches to software engineering: programming methodology, best practice, CASE tools, etc., are only *ways of better organizing the handcraft*, akin to giving foot soldiers a better training. It is (at least to me) clear that *automatic* and *mechanical* treatment of programs is needed to lift software development above its current state.

The promise of PE is just this: its ability to lift parts of program development to a highly automated, near-industrial process, where human creativity and even *human comprehension of individual programs* is only occasionally required: when high level computational decisions are to be taken.

1.1 The roots of partial evaluation

Partial evaluation is at the meeting point of three areas, all dealing with programming languages but in rather different ways: *compilation*, *semantics*, and *computability theory*.

PE techniques extend compilation technology in both practice and theory. Compiling concepts that overlap heavily with PE include *procedure cloning*, *inlining*, *loop unrolling*, and *constant propagation*. Program *flow analysis*, or its semantics-based counterpart *abstract interpretation*, is essential; PE systems do much flow analysis, either online during specialization, or offline during a program preprocessing phase. In principle a partial evaluator can be regarded as an unusually aggressive compiler; but its applications reach far beyond those considered normal even for highly optimizing compilers.

The view of programs as data objects requires a precise knowledge of *program semantics* for correct transformation and analysis so semantics, formal or informal, is the foundation of PE. Here *operational semantics* seems to be the best medium for formulations and correctness proofs[1].

PE treats *programs as data objects* that can be manipulated like any other data, and which in addition can be run, constructed, decomposed, or transformed. This line of thought stems from the 1930s and 1940s(!) when *recursive function theory* first bloomed. Researchers then explicated the meaning of recursive definitions, introduced the first self-interpreters ("universal machines"), and used self-application for undecidability of the *halting problem* and Kleene's *second recursion theorem*. Further, PE is "just" an efficient implementation of Kleene's *S-m-n theorem*.

In the rest of this article a program is *always a textual object*: a view entirely compatible with both recursive functions and compilation, but in contrast with, say, the λ-calculus, where a program is an equivalence class under

[1] A problem with denotational models using reflexive function domains is that most lack "full abstraction," meaning that they cannot adequately formalize programs' operational behavior. A consequence: such models are too conservative when replacing program fragments by equivalent ones.

$\alpha\beta\eta$-convertibility. Further, *language definitions* and other *problem specifications* will also be regarded as concrete data objects.

1.2 Some notations to describe PE

We will write $[\![\mathtt{p}]\!]$ for the *meaning* of `p`, which in this paper is always a function with first-order inputs and outputs. If we wish to specify the language `L` in which `p` is written, we will write $[\![\mathtt{p}]\!]^{\mathtt{L}}$. The output resulting from running `p` on input `d` will be written as $[\![\mathtt{p}]\!](\mathtt{d})$ or $[\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{d})$.

Manipulation of multistage programs such as compilers, interpreters, and specializers requires precise notations for the effect of program runs and the correctness of the manipulations. Runs are described by *expressions*, e.g. $[\![\mathtt{p}]\!](\mathtt{d})$ describes a one-step run, and $[\![[\![\mathtt{compiler}]\!](\mathtt{source})]\!]^{\mathtt{T}}(\mathtt{d})$ describes a two-step run that first compiles program `source` and then runs the resulting target program on input `d`.

Equalities between such expressions describe criteria for *correctness*, e.g. a target program is correct with respect to a source program if

$$[\![\mathtt{source}]\!]^{\mathtt{SourceLang.}}(\mathtt{d}) = [\![\mathtt{target}]\!]^{\mathtt{TargetLang.}}(\mathtt{d})$$

for any input `d`. (Here "equality" must include the possiblilty that one side is undefined, in which case the other side should also be undefined.) Figure 1.1 defines in this way the meaning of *target program*, *interpreter*, and *compiler*.

Equational specifications give *clear and precise statements of goals*; and equalities between the expressions allow *reasoning*: to predict in advance the net result of certain runs, even though they may be hard to grasp intuitively in terms of program execution. (A notorious but familiar example is keeping track of program versions during "bootstrapping".)

A *correct partial evaluator* (generically called `mix`) is easy to define equationally. For any program `p` expecting two inputs `s` and `d`, program `mix` must satisfy:

$$[\![\mathtt{p}]\!](\mathtt{s},\mathtt{d}) = [\![[\![\mathtt{mix}]\!](\mathtt{p},\ \mathtt{s})]\!](\mathtt{d})$$

The name "`mix`" for a partial evaluator was coined by Ershov to express the fact that any efficient PE must do a *mixture* of execution actions (those of `p`'s actions that depend only on `s`), and compiling actions for the remainder. The program output by `mix` was appropriately called the *residual program*.

1.3 Two results predicted by equational reasoning

Equations can, as mentioned, *define the desired net results* of computer runs whose time or space efficiency are hard to anticipate in practice. This is a virtue, as it makes it possible in advance to state WHAT one wishes to achieve before going into operational detail as to HOW. Further, it gives a way to check *whether* a goal was in fact achieved (i.e. perform the runs and compare the results), before going into details as to *how well* it was done.

This idea is so familiar as to be banal in mathematics; one instance is to find roots to an algebraic equation. Nonetheless symbolic and algebraic thought patterns are not yet as natural in programming languages as they could and should be.

$$\begin{aligned}
\mathtt{output} &= [\![\mathtt{source}]\!]^{\mathtt{SourceLanguage}}(\mathtt{input}) \\
&= [\![\mathtt{target}]\!]^{\mathtt{TargetLanguage}}(\mathtt{input}) \\
&= [\![\mathtt{interp}]\!]^{\mathtt{ImplemLanguage}}(\mathtt{source},\mathtt{input}) \\
\mathtt{target} &= [\![\mathtt{compiler}]\!](\mathtt{source}) \\
\mathtt{compiler} &= [\![\mathtt{cogen}]\!](\mathtt{int})
\end{aligned}$$

Figure 1.1: Interpreter, target, compiler, compiler generator

$$\begin{aligned}
\mathtt{target} &= [\![\mathtt{mix}]\!](\mathtt{int},\ \mathtt{source}) \\
\mathtt{compiler} &= [\![\mathtt{mix}]\!](\mathtt{mix},\ \mathtt{int}) \\
\mathtt{cogen} &= [\![\mathtt{mix}]\!](\mathtt{mix},\ \mathtt{mix})
\end{aligned}$$

Figure 1.2: The Futamura Projections

Two good examples are the *Futamura projections* (explained in the next section) and the *Ceres* compiler generator system [11, 29]. In each case, certain equations, if solvable, implied the existence of programs with some desirable properties (in these cases, two compiler generators).

Further, in each case one had *no intuitive idea* of what the programs satisfying the given equational specifications would look like, or what their efficiency would be (if indeed they existed at all)[2]. In fact, and to our surprise, the compilers and the compiler generator produced by Ceres turned out to be reasonably efficient.

A limitation was that Ceres only automated the *code generation* task, and took no account of other statically performable actions such as looking variables up in a symbol table. Analogously, Dave Schmidt and I saw the need to *separate the binding times* of static and dynamic parts of computations in a λ-calculus-based approach to compiler generation (the Aarhus *Semantics-Directed Compiler Generation* workshop in 1980 [17]), but it was quite unclear how to do it. Partial evaluation gave an answer, four years later.

1.4 Self-application

The idea that self-application of a program specializer could lead to automatic program generation was understood independently in Japan by Futamura; in the USSR by Turchin and Ershov; and in Sweden (where I first heard the idea, in 1981) by Sandewall's group. The most precise and concise expression of this idea is through the Futamura projections of Figure 1.2. A consequence, easy to verify from the definitions of `cogen` and `mix`, is that `cogen` = $[\![\mathtt{mix}]\!](\mathtt{mix},\ \mathtt{mix})$

The generating extension. Claim: any two-input program `p` has a corresponding "generating extension" `p-gen` (term due to Ershov) with the property that

$$[\![\mathtt{p\text{-}gen}]\!](\mathtt{s}) = [\![\mathtt{mix}]\!](\mathtt{p},\mathtt{s})$$

for any "static" (known first) data value `s`. In words, `p-gen` is a *generator of specialized versions of* `p`. When given `s`, pro-

[2] Remark: just as some equations have no roots, one can write equalities describing unachievable program behavior.

gram `p-gen` directly generates a specialized version without having to run the general program `mix`.

How do we know such a program exists? Easy algebra verifies that $[\![\texttt{mix}]\!](\texttt{mix, p}) = [\![\texttt{cogen}]\!](\texttt{p})$, and that the program both define satisfies the equation above. Consequently a generating extension `p-gen` may be built by either of the two runs (the one involving `cogen` is faster, though).

This generating extension is a specialized version of `mix`, so it can be expected to, and does, run significantly faster than `mix` itself. Fast specialization has an exciting potential application for *runtime code generation*, e.g. for use in an operating system (Consel, Lee, Massalin, Pu).

When a user asks one to specialize a given program `p`, one may deliver its generating extension `p-gen`. This has two consequences. First, the user can create as many specialized versions of `p` as desired simply by running `p-gen`; and he or she need not know how to use the perhaps complex partial evaluator `mix`. A second effect is *security*: there is little risk that `p-gen` could be "reverse engineered" to reconstruct the partial evaluator itself. A fruitful application of this concept, *not* involving self-pplication, will be seen later.

2 Origins of partial evaluation

Early times. PE has been independently discovered many times, perhaps first by Kleene in computability theory, and many times since Lisp and Markov algorithms were devised. Early researchers include Lombardi and Goad in California, Futamura in Japan, Turchin and Ershov in the USSR, and Sandewall's group in Sweden. Some of their insights were purely theoretical, e.g. Kleene's construction actually gave program slowdown rather than speedup. Early practical work was more or less ad hoc with unpredictable results. Such weaknesses are completely understandable, as no formal semantics existed at the time, so correctness criteria were hard to express (an exception is Goad's work based on pruning proof trees).

Yoshihiki Futamura discovered the potential of self-application of PE for compiling and compiler generation in 1971. Independently, Valentin Turchin pursued the topic energetically in the USSR in the early and mid 1970s, building a research group in Moscow which is still active in the field. Andrei P. Ershov made a major contribution by a series of lectures in the West, mostly in the 1970s, clarifying concepts and emphasizing the way PE unifies the concepts of interpretation, compilation, and program optimization.

The dream of compiler generation from denotational semantics. After learning denotational semantics, I discovered from Peter Mosses' SIS project that denotational semantics definitions were executable (by doing λ-reduction). It occurred to me that since enough information is present to *execute* language definitions given a source program and its input data, there must be information enough to *compile* that source program into a target program. This dream led to an early conference, *Semantics-Directed Compiler Generation* [16] in Aarhus in 1980.

Gaining high efficiency while staying faithful to purely denotational semantics turned out to be quite difficult, partly for domain-theoretic reasons. The goal of true semantics-directed compiler generation remained elusive for several years, as the approaches tried by several researchers[3] were too slow, too limited in power, too complex, too hard to automate, or not solidly semantics-based. Much of this later led to more sophisticated work; for example many of Appel's thesis ideas have been industrially realized in the SML-NJ compiler. Better techniques to separate *compile-time* from *run-time* actions were commonly seen as necessary, but it was unclear how to do it in full generality.

Several years later the dream was fulfilled using PE: a successful experiment by Jørgensen [23] generated from a purely denotational definition of a lazy functional language, similar in spirit to the Miranda^{TM} language, a compiler yielding faster target program run times than Turner's commercial system. Unfortunately, by that time there was less interest in purely denotational language definitions.

Deficiencies of denotational semantics as a tool for language definitions (for example problems with concurrency, the full abstraction problem, and lack of readability), led to alternative language definition formalisms, one example being Mosses' "Action Semantics." Experiments in its implementation, using partial evaluation and abstract interpretation, have been done by Bondorf, Palsberg, and Ørbæk.

First steps towards self-applicable PE. The partial evaluators implemented by Sandewall's group in Sweden dealt with full Lisp, "warts and all," and were not self-applicable. An important step was their development of REDFUN, which can be thought of as a handwritten version of `cogen`.

A paper by Ershov asserted that compilers could be generated by self-application of a PE, but didn't say whether this had been achieved on the computer. Ershov told me at IFIP 1983 in Paris that it had not. An at first purely intellectual question arose: could the Futamura projections be realized *by any nontrivial specializer at all*[4]?

3 Developments 1984-1995

3.1 Conceptual understanding: online and offline PE

Kleene's construction of $\texttt{p}_\texttt{s} = [\![\texttt{mix}]\!](\texttt{p, s})$ was simple: $\texttt{p}_\texttt{s}$ was identical to `p`, except for code to initialize its first input argument to `s`. Less trivial PE requires specialization-time operations (unfolding, symbolic computation, etc.) of unbounded length. Their task is to perform certain (not necessarily all) of `p`'s *static* operations or commands: those which depend only upon `s`; and to *suspend*,i.e. to generate code for, the remaining ones, called *dynamic*.

Common problems to be overcome include sufficiently strong specialization, ensuring that specialization will terminate, and dealing with data containing both static and dynamic components, or static components built up under dynamic control. The problem with the latter is that `mix` has to deal with all possible run-time execution patterns, so specialization can loop while trying to account for infinitely many different run-time computations, *even though every single computation terminates*

Individual acts of unfolding and symbolic computation are simple and obviously semantics-preserving; the main problem is controlling their repeated use. Of the two major

[3] These included Appel, Christiansen, Gaudel, Lee, Mosses, myself, Paulson, Pleban, Raskowsky, Tofte, Vickers, Wand, and Weis.

[4] Kleene's S-m-n theorem gives a true solution, but one that is useless practice.

schools of thought as to how to do this, the oldest is *online* specialization, in which p's execution is mimicked directly, and decisions are taken on the fly as to which of of p's operations to suspend. In comparison, *offline* specialization begins with a preanalysis of the program (in which it is only known *which* inputs will be known during specialization but not what their values are), during which decisions are made in advance as to what to simulate, and what to suspend.

Most online methods have sophisticated *loop detectors* applied at PE time; whereas offline methods perform sophisticated *program analyses* during preprocessing. Both work by forms of *abstract interpretation*, manipulating abstract descriptions of sets of possible runtime computational states; and achieve termination by *generalization*: replacing one state set description by another less precise one which describes a larger set of states.

Online PE. The first partial evaluators were online, and this approach is still very useful. Online specialization can seize more opportunities for code improvement so as to squeeze the last microseconds of performance out. Further, this approach naturally allows *partially static data structures* and *polyvariance* in binding times. (The latter means, for instance, that a function of say two arguments can give rise to specialized versions in which the first is known, or the second, or neither, or both).

The Swedish systems developed by Sandewall's group for Lisp and Prolog were online. Newer systems (named later) use rather more advanced techniques to ensure termination.

Offline PE. The motivation for the first offline PE systems was to get self-application to work (see the next section). Since that time, several other good reasons have been discovered favoring offline PE. However the choice between online and offline is not black-and-white, and some applications of PE favor the one approach and some, the other. Further, larger systems usually have a measure of both, e.g. the Similix system has extensive offline analyses, but also a postprocessor for final "tidying up" which is essentially a simple online PE pass.

Offline systens have a preprocessing phase, often called *binding-time analysis* (BTA for short) with program analyses ranging from a simple *dependency analysis* to see which value cannot be computed without knowing p's dynamic argument, to a spectrum of other analyses to ensure good behavior during specialization. (Several are mentioned below.)

The output of preprocessing is typically an *annotated program* $\mathtt{p}^{ann}$, identical to p but with marks placed that in effect say "perform this operation during specialization," or "generate code for this operation" or "generate a specialized program point if control reaches here." The specializer proper can then be small and fast, as its task is just to obey the annotations without having to take decisions online.

A benefit of this approach for those unfamiliar with the mysteries of PE is that the annotated program $\mathtt{p}^{ann}$ gives the user feedback *in terms of user's own program*, so there is little need to know how mix works.

Binding-time improvements. What to do if $\mathtt{p}_\mathtt{s}$ shows disappointingly small speedup, or is execessively large? A *binding-time improvement* is a modification of an already working program, so as to obtain better results from partial evaluation. With offline methods, the user has something to study: his or her annotated program $\mathtt{p}^{ann}$, and p can be tweaked so as to get a $\mathtt{p}^{ann}$ with more numerous static computations. (Note the analogy to tuning a grammar to make it acceptable to a parser generator such as Yacc.)

Further, annotations make it easier to reason about problems of code explosion and nontemination of specialization; with online methods the first resort (not easy) is to examine the output of mix to see what went wrong, and what to do next; the other resort is unreasonable or any but the most hardcore users: to *study the behavior and structure of* mix *itself.*

3.2 Achievement of self-application

Experiments at DIKU began with Peter Sestoft, Harald Søndergaard, and other students in late 1983, to construct a self-applicable mix; and in parallel for a few months I tried (seriously!) to prove that any self-applicable program specializer must be trivial. After more than a half year of frustrating experiments, our first (enormous but correct) generating extension was produced: a 3-line program p yielded a p-gen = [[mix]](mix, p) which worked, but was 500 pages in length(!); and an acceptably efficient fully self-applicable mix was first produced in November 1984.

It took some time to understand that [[mix]](mix, p) was huge because it was *overly general*; most of the code was determining whether or not it should do a computation online or generate code instead, in situations where it was perfectly clear (to us) which choice should be taken. Our first solution to this problem was a "hack": hand addition of ad hoc annotations to p, to tell the second mix which of p's operations to simulate and which to generate code for.

This gave an enormous reduction in the size of p-gen. A pattern was seen which often holds today: a mix-generated compiler is often about 3 times larger than the interpreter from which it was derived; and running a mix-generated target program is often about 10 times faster than running the interpreter. In 1984 we began to understand that the annotations in essence carried information about the *times* at which various parts of p (and mix itself) could be performed; first automated the annotation process; and coined the term "binding-time analysis" for it.

Biasing the inputs. In general form the second Futamura projection is p-gen = [[mix]](mix, p). Without binding-time information of some sort, the *second* mix *has no information* that p is *to be specialized to its first argument* and not, for example, to its second. An unannotated mix thus has to take account of both possibilities.

W saw that in case p is an interpreter, p-gen is a compiler. Without annotations or similar information, the result will in fact be more: a program tailored to specialize p, a special case of which is the task of compiling a source program. Seen as a compiler, it is most peculiar and inefficient, as it is able not only to compile by specializing an interpreter to its program input, but it can also specialize the interpreter to its second (runtime data) input *when its program input is unknown.*

Using annotations implies the generating extension is p-gen = [[mix]]($\mathtt{mix}^{ann}$, $\mathtt{p}^{ann}$) instead of p-gen = [[mix]](mix, p). The role of annotation is thus to indicate a binding-time *bias*: that program p-gen will be applied to the value of p's first input, and not its second.

3.3 Technical advances

Rapid activity has occurred at many locations in the past decade. Following is a selection of advances I found particularly significant (with apologies to anyone whose work may have been indavertently omitted).

Partial evaluation systems. These include **C-mix**, for all strictly conforming ANSI standard C programs; **Fuse**, for Scheme; **Mixtus**, for essentially all of Prolog; **Sage**, for the strongly styped logic programming language Gödel; **Schism** and **Similix**, for Scheme; and **Unmix**, for a first-order subset of Lisp. More experimental systems include **Fspec** for a subset of Fortran 77; **PEL**, for a monomorphic typed first-order functional language; **λ-mix**, for the untyped lambda calculus with constants; **Potyful**, for a polymorphic first-order functional language; **SML-mix**, for a subset of Standard ML; and the **supercompiler** for Refal. Most are *offline*, exceptions being the supercompiler, Fuse, and Mixtus.

Self-applicable systems include λ-mix, Sage, Similix, Schism, and Unmix; and recently a restricted supercompiler.

The Moscow Unmix system is a particularly elegant, simple and cleanly written example, good for study by newcomers. It follows the lines of the original system (no longer available), and has an "arity raising" postprocessor that makes specialized programs much more readable.

Similix is DIKU's most complete system and is publically available with user manual, advice on and user feedback to aid binding-time improvements, etc. It has now been copied by FTP to several hundred sites.

Credits: C-mix, λ-mix, Similix, and SML-mix at DIKU by L O. Andersen; Carsten Gomard and myself; Anders Bondorf and Olivier Danvy; and Lars Birkedal and Morten Welinder. Fuse by Dan Weise and students at MIT, Stanford, and Microsoft in Oregon. Mixtus at SICS in Sweden by Dan Sahlin; Sage at Bristol by Corin Gurr. Schism by Consel and Danvy in France (Paris, Rennes) and Oregon (OGI). Unmix at Moscow by Sergei Romanenko. Fspec at Vienna by Robert Glück and students). PEL at Glasgow by John Launchbury. Potyful at Leuven by Anne De Niel and Dirk Dussart.

Partially static data structures The online systems Fuse, Mixtus, and Turchin's supercompiler handle partially static structures quite naturally. Further, SML-mix, Sage, PEL, and Potyful can handle partially static structures by virtue of their type systems.

Torben Mogensen did the first implementation of an offline PE system able to handle data structures that are partially static and partially dynamic. His approach was based on *tree grammars*. Since then this ability has been added to Similix and Schism, both using tables binding information to data structure "creation points" — in essence another way to repreent a grammar.

Higher-order functions It was several years before extension beyond first order could be done; and the breakthrough happened nearly simultaneously , with λ-mix (Gomard and Jones at DIKU) for the λ-calculus with constants and Similix for Scheme (Bondorf, while visiting Dortmund, and Danvy). Schism was extended to higher-order functions not long after (Consel and Danvy). A novel form of binding-time improvement can be achieved in these systems, by converting programs to *continuation-passing style*. The point is that such programs often have better binding-time separation than the programs they were derived from.

Preservation of termination properties and side effects. Similix is the only system of which I am aware that has careful analyses to detect both expressions that might cause side effects, and ones that might be duplicated or eliminated during specialization. Similix is very conservative in such cases, to avoid causing any change whatsoever in program semantics by specialization. One could maintain that it is not dangerous to reduce `(car (cons E1 E2))` to `E1`. However, semantic changes can happen, for example if `E2` loops but `E1` does not (increased termination, not so bad), or if `E2` can commit an error or write on an output file (more serious).

Code size explosion. The same analysis also reduces (but does not eliminate entirely) the risk of code size explosion. The reason is that, to avoid repeated side effects (especially undesirable if input-output are involved), Similix inserts `let` expressions so the computations involved are only performed once. This also ensures that the code only appears once in the specialized program. Other problems with code size explosion will be discussed later.

Imperative languages The C-mix and Fspec systems handle explicitly inperative languages, and even Similix allows side effects. Imperative languages are much trickier to specialize than functional or logic lnguages. One reason is that imperative languages allow access to *all* variables in the current scope, with the reult that code explosion can occur unless care is taken *not to specialize with repect to dead static variable values.*

Another is that a PE system must account for *all possible* run-time states, and the effects of assignment can cause nonlocal effects that must be accounted for. In particular, the C language requires a *pointer analysis*, since assignment of a dynamic value through one pointer variable also affects any others with which it may be aliased.

Binding-time improvements There is a growing compendium of experience in this subtle art. The Similix manual has a section on binding-time improvements, and several techniques are described in [20].

3.4 Staging levels and evaluation times

Multistage program runs, e.g. bootstrapping and selfapplication are confusing. To keep language levels unambiguous we always use the semantic function $[\![]\!]$ to make program runs explicit, e.g. we write $[\![\mathtt{p}]\!](\mathtt{d})$ and never expressions such as `p(p)` which implicitly assume the first `p` is to be executed.

A rudimentary type system can explicitly express the notion of "staging level" and so help keep track of levels. Recall that *all* programs are textual objects, so there is no need for Gödel numbers or other indirect program representations. The underlying idea is that *values* have types (e.g. integer, Boolean), whereas *programs* have *program types* that explicitly give the language used to interpret them. For simplicity, and to avoid the need for type coercion functions, we assume all first-order values can be expressed in a common format and ignore encodings of values from one domain into another. This assumption will be re-examined later.

Underbar types Keeping levels distinct implies a change in conventional notations. As an example, we would not write `f(3)+1:int`, but for example `f(3)+1:`$\underline{\texttt{int}}_{ML}$, meaning "expression `f(3)+1` is an expression which, if evaluated by language ML, would yield a value of type `int`." We define the meaning of type expression $\underline{t}_{\texttt{L}}$, where `L` is a programming language, to be the set

$$[\![\underline{t}_{\texttt{L}}]\!] = \{\texttt{L}-\textit{program}\ \texttt{p} \mid [\![\texttt{p}]\!]^{\texttt{L}} \in [\![\texttt{t}]\!]\}$$

A compiler should translate a source program into a target language program whose meaning is identical to that of its source program. In other words a compiler is a meaning-preserving program transformation, insensitive to the type of its input program. Thus the identity function, with type $\underline{t}_{\texttt{L}} \to \underline{t}_{\texttt{L}}$ for all types t, is a trivial but well-typed compiling function from `L` to `L`. The types are polymorphic in that t can range over all first-order types.

Given a source `S`-program *that denotes a value of type* t, an `S`-interpreter (when run) should return a value of type t. A partial evaluator has two inputs: a program, and a value of its first input. For convenience we use "curried" notation, writing $[\![\texttt{p}]\!](\texttt{s})(\texttt{d})$ and not $[\![\texttt{mix}]\!](\texttt{p, s})$.

In the following, `S` is the input language to a interpreter and compiler, and `T` is the compiler's target language. When describing partial evaluation, we omit the subscript `L`. The types assigned above to interpreters etc. assume that all observable values are `S`-, `T`- and `L`-types.

$[\![\texttt{interpreter}]\!]$: $\underline{t}_{\texttt{S}} \to t$ and

$[\![\texttt{compiler}]\!]$: $\underline{t}_{\texttt{S}} \to \underline{t}_{\texttt{T}}$

In comparison with:

$[\![\texttt{mix}]\!]$: $\underline{t \to t'} \to t \to \underline{t'}$ and

$[\![\texttt{cogen}]\!]$: $\underline{t \to t'} \to \underline{t \to t'}$

Remarks. The different output types of a compiler and an interpreter are quite natural. Note that $[\![\texttt{interpreter}]\!]$ illustrates dependent types in their most extreme manifestation, since the value produced by $[\![\texttt{interpreter}]\!]^{\texttt{L}}(\texttt{s}\mathit{ource}, \mathit{data})$ can be *any type that can be expressed by any* `L`*-program.*

Consider the simple type deduction rule: if `p`: $\underline{t}_{\texttt{L}}$, then $[\![\texttt{p}]\!]^{\texttt{L}}$: t. If we assume $[\![\texttt{p}]\!]$: $t \to t'$ and `s`: t, we can conclude from the types of `mix` and `cogen` above that $[\![\texttt{mix}]\!](\texttt{p})(\texttt{s})$: $\underline{t'}$ and $[\![\texttt{p-gen}]\!]$: $t \to \underline{t'}$. This rudimentary type system is sufficient to account for self-application; all the Futamura projections can be shown to preserve the expected types in this notation. For details, see [20].

Writing `cogen` instead of `mix`. Several advantages accrue from writing `cogen` instead of `mix`, especially when dealing with typed languages. The reason can be seen by comparing the level types of `mix` and `cogen` above; the latter is merely a *code-to-code transformer*, whereas `mix` must deal with input values of type t and program texts of type $\underline{t \to t'}$ at the same time.

Two partial evaluators at DIKU have been written this way. An earlier version of C-mix had the `mix` type above, but has now been revised to have the `cogen` type; and SML-mix was written that way from the beginning. The net result for both Similix and C-mix was to enlarge the range of language features that could be handled, and to increase their efficiency during specialization by quite substantial amounts.

4 Assessment

In my opinion major progress on technical problems has been made. Still, much remains to be done, partly concerning typed languages,and especially as concerns making PE usable by others than ourselves. The past decade began with the puzzle of achieving self-application. This has now been achieved with full success, even with respect to an objective criterion for "optimality" explained below.

4.1 New insights

A basic conflict: computational completeness versus termination. It is clearly desirable that specialization function $[\![\texttt{mix}]\!]$ is *total*, so every program `p` and partial input `s` leads to a defined output $\texttt{p}_\texttt{s} = [\![\texttt{mix}]\!](\texttt{p, s})$. On the other hand, a demand for *computational completeness* is equally natural: given program `p` and partial data `s`, *all* of `p`'s computations that depend only on input `s` will be performed. Most online PEs favor totality, while offline ones tend to favor completeness.

Unfortunately these two are in fundamental conflict. If, for example, `p`'s computations are independent of its second input `d` and $[\![\texttt{p}]\!]$ is a partial function, then computational completeness would require $[\![\texttt{mix}]\!](\texttt{p.s})$ also to fail to terminate whenever $[\![\texttt{p}]\!](\texttt{s}, \texttt{d}) = \bot$.

A tempting way out is to allow $\texttt{p}_\texttt{s}$ to be less completely specialized in the case that $[\![\texttt{p}]\!](\texttt{s, d}) = \bot$, e.g. to produce a trivial specialization. This is, however, impossible in full generality, as it would require solving the halting problem.

Online specializers typically monitor the static computations as they are being performed, and force less thorough specialization when risk of nontermination is detected. Such a strategies, if capable of detecting *all nontermination*, must necessarily be less than complete in some cases, for if perfect they would have solved the halting problem.

A typical example hard to specialize nontrivially without having the specializer fail to terminate:

```
if complex-but-always-true-condition-with-d
then X := 'nil
else while 'true do S := cons S S;
```

One cannot reasonably expect the specializer to determine whether the condition will always be true. A specializer aiming at computational completeness will likely attempt to specialize both branches of the `while` loop, leading to nontermination at specialization time.

Optimal interpreter specialization. A specializer should be 'optimal' when used for compiling, meaning: it removes *all interpretational overhead.* This can be made more precise given a self-interpreter `sint`. By the first Futamura projection, for every input `d`

$$[\![\texttt{p}]\!](\texttt{d}) = [\![\texttt{sint}_\texttt{p}]\!](\texttt{d})$$

where $\texttt{sint}_\texttt{p} = [\![\texttt{mix}]\!](\texttt{sint})\texttt{p}$, so program $\texttt{sint}_\texttt{p}$ is semantically equivalent to `p`. One could reasonably say that the specializer has *removed all interpretational overhead* if $\texttt{sint}_\texttt{p}$ is at least as efficient as `p`.

The concept of 'optimality' has proven itself very useful in constructing practical specializers, and for several (at least Unmix and λ-mix) the specialized program $\texttt{sint}_\texttt{p}$ is

identical up to variable renaming with program `p`. Further, achieving optimality has shown itself to be an excellent stepping stone towards achieving satisfactory compiler generation by self-application.

Linear speedup is the rule. All PE systems of which I know in effect simulate the program to be specialized in its same computation order, with the consequence that computations by the specialized program $\mathtt{p}_\mathtt{s}$ on input `d` *can be embedded, 1-1 and in the same order* in the computations of `p` on input `s, d`.

Suppose one, for a fixed `s`, compares the run times of `p` and $\mathtt{p}_\mathtt{s}$ on an input `d` which grows towards infinity. A consequence of the previous point is that *superlinear speedup cannot occur,* since the computations "saved,"i.e. precomputed by `mix`, must necessarily be a function of the static data `s` alone.

Two ameliorating factors: first, the speedup is nonetheless often large enough to be of major practical significance. Second, the coefficient may depend on the static data, so *larger values of* `s` *can result in larger speedups.*

Finally, even though complexity theorists have told us for years that "constant factors are trivial," I proved in [21] that this is in fact *not true* for a more realistic computation model than the Turing machine. Interestingly, the insight that led to this result came as a result of explaining one of our self-interpreters to a visiting complexity theorist.

Specialized program points by means of projections. An early insight into the underlying nature of program specialization was to see that most existing systems worked by *projections*: given a total computational state or configuration, `mix` proceeded by throwing away the part of it depending on dynamic input, and maintaining an abstract description of the remaining, static part. This was formalized in [19], and John Launchbury later extended the ideas to typed programming languages, using *domain projections*, in his Ph.D. thesis [24].

Specialized program points by means of abstract interpretation. Robert Glück and Andrei Klimov clarified in [12] what Turchin had said earlier: that supercompilation could achieve degrees of specialization beyond what could be done by projections alone. It became clear that what was happening was a kind of *online abstract interpretation* of the "relational" rather than "independent attribute" sort, and this was formulated in language-independent terms in [22].

Code size explosion. PE sometimes gives rise to specialized programs too large to be of practical use (though it rarely happens when compiling by specalizing interpreters). Insight has been gained into this problem, and parts have been solved, e.g. by the code duplication analysis which is now part of Similix.

Another source of code explosion is the presence of *functionally independent static variables.* Since `mix` has to account for *all possible run-time scenarios*, specialization of, say, a function with several independent static arguments can lead to very many specialized versions of this function.

Yet another problem, especially for imperative languages, is that of *dead static data.* The problem here is that imperative programs always have the possibility of access to all variables whose scope includes the current control point. If some of these are static and as well "dead" (i.e. will never be used or tested upon again), they can, just as in the previous point, give rise to numerous specialized program points. In contrast, though, all these specialized program points are *semanticaly equivalent* and so the code duplication is completely without utility.

A single universal data type, versus one type universe for each type. Strongly typed programming languages are popular, and have many advantages for human programmers. The ideas involved in PE can be extended to first-order typed values, e.g. as in the thesis work of John Launchbury at Glasgow and in SML-mix (Birkedal, Welinder). Extending the concepts to types creates some difficulties, though, partly when attempting to compile by specializing interpreters, and especially when attempting self-application.

The Futamura projections and other equations above all assume a "universal" data type containing all first-order values, including the texts of programs. For example the definition of "interpreter" states that its output is *identical* to that of the program it is simulating, whatever its type may be.

In a typed context, where every type definition defines its particular domain of discourse (i.e. the "Church" view rather than the "Curry" view of types), interpretation of typed programs would necessarily involve some form of coercions to and from the interpreter's input type, for example:

$$\mathit{encode} : \mathit{ArithmeticExpressions} \rightarrow \mathit{LispLists}$$
$$\mathit{decode} : \mathit{LispLists} \rightarrow \mathit{ArithmeticExpressions}$$ and

In the world of typed partial evaluation one often sees expressions such as $\overline{exp}^{Lisplists}$, standing for "a representation of arithmetic expression *exp*, encoded as a Lisp list." Time and space overhead: usually a constant factor, and can often be quite large.

This pattern *cannot*, however, be extended to domains containing functions. It does work in one direction; the following is used quite often:

$$[\![\mathtt{p}]\!]: \ \mathtt{L}\text{-}\mathit{programs} \rightarrow (\mathtt{LispLists} \rightarrow \mathtt{LispLists})$$

but there is no natural inverse:

$$[\![\mathtt{p}]\!]^{-1}: (\mathtt{LispLists} \rightarrow \mathtt{LispLists}) \rightarrow \mathtt{L}\text{-}\mathit{programs}$$

One problem is that $[\![\mathtt{p}]\!]^{-1}(f)$ will not exist at all in case f is not a computable function. Even if f is computable, the notation is dangerously ambiguous since there are infinitely many different programs to compute the same computable function, and these may vary widely with respect to run time — the very factor that one wishes to optimize by PE.

By a lucky accident the first partial evaluators were all written for untyped languages, as practitioners were unaware of these problems at that time. We escaped these problems above by using one universal data type for first-order data (e.g. Lisp lists), avoiding the need for functions such as *encode* and *decode*.

Problems with specializing strongly typed languages. Program `mix` is, as is clear from its underbar type, part interpreter and part compiler. The interpretive component is especially problematical for strongly typed

languages. Such an interpreter must use some form of "universal type" abe to encode *all values* manipulable by any program being interpreted, for example $V = Int + Bool + V \times V + V \to V + \ldots$.

The problem is that specialized versions of such an interpreter naturally "inherits" the universal type, so *every single run-time value* involves type tags associated with V. Experiments reveal this to have unexpectedly high costs, both in terms of run-time storage, and time to set, test, and remove the (almost all unnecessary) run-time type tags.

How can this overhead be removed? Reasonably efficient special solutions have been devised at specialization time (Launchbury, De Niel), but target programs remain quite unnecessarily slow and large.

The obvious general answer is some form of *type specialization*, but it is as yet unclear how to formulate this concept, let alone how to solve it. Researchers who have worked on the problem or are now doing so include Launchbury, De Niel, Dussart, Mogensen, and Welinder.

4.2 Advantages of writing `cogen` instead of `mix`

For the reasons just given, a partial evaluator `mix` for a strongly typed language will almost certainly be inefficient at specialization time due to the need for a universal type, and compilers generated by self-application will both be slow and generate slow target code.

Fortunately there is a way out: to write `cogen` instead of writing `mix`. Its input is an annotated program $\mathtt{p}^{ann}$, and its output is a generating extension `p-gen`, able to produce specialized versions of `p`. Examining the underbar type of `cogen` reveals that it does not deal with both values and program texts, as do interpreters and `mix`. Program `cogen` is simply a text transformer.

Somewhat oversimplified: its task is just to generate code to appear in `p-gen` from the statically annotated parts of $\mathtt{p}^{ann}$; and to generate code generation instructions from the dynamically annotated parts of $\mathtt{p}^{ann}$. The code part of `p-gen` can simply be a copy of the static parts of $\mathtt{p}^{ann}$, *including all its static type declarations*, so there is no need for static computations to be performed interpretively.

Side effects of this approach include greater efficiency at specialization time since static computations are performed directly and not interpretively; and ease of handling semantically complex language constructions, as these are just "deported" to either `p-gen` or the specialized version of `p`. *Warning*: for correctness, on must verify (if true) that the meaing of such semantically complex constructions will be preserved under unfolding, etc. as done during specialization.

5 What next?

What should we do next in the field of partial evaluation? One answer is to expand horizons: to solve new problems, to build new systems with greater efficiency, handling more powerful languages or able to handle new applications. Such improvements are clearly desirable in the long run, but there is a danger of getting more and larger programs, but of failing to gain new insights usable by others.

A second is to understand in precise terms *just what it is that our methods and our programs accomplish*, to evaluate them, and to find out *what their inbuilt limitations* are. This understanding is essential for two very practical reasons:

- Communication of results so they can be adapted to solve related problems.
- Achievement of greater automation in program handling. Ideally our program transformation tools should be *anonymous parts of a program management system*, as invisible to the user as gears or transistors — and this requires a precise knowledge of their characteristics.

5.1 General directions for future development

A quite remarkable impact on software development: automatic specialization tools allow the design of reusable software that is general, well-structured and easier to maintain, but without the penalty of being too inefficient. This is not a new idea; for example macro processors have long been used to customize, say, operating system kernels to particular hardware configurations. Further, the ideas of "shrinkwrapped" software or "procedure cloning" involve the same concepts.

Use of PE to aid software maintenance and reuse. Reuse can be achieved using a different view of specialization: given a program `p`, first (by hand) generalize it to an another more generic program `g`, e.g. by adding parameters to give it a *broader functionality*. Then program `g` can automatically be specialized with respect to different parameter settings. Further, there is a simple test for verifying the correctness of the generalization: specialize `g` to the parameters defining `p`'s original problem and see if the result is close to `p`.

Making PE usable by nonexperts. This goal invoves several requirements. One is to discover *which types of problems* yield good specialization, and another, *which environmental support tools are needed* for effective use of a specializer by nonexperts. It is important to identify trouble spots where good specialization cannot be achieved, or where it can be achieved but only by methods hard to learn by industrial users.

Binding-time improvements. Practioners in the PE field have a growing accumulation of experience; it is strongly desirable to consolidate this experience into a publically accessible compendium.

5.2 Needed to achieve wider acceptance of PE.

Wider use of partial evaluation requires abandoning any remaining "black magician" attitudes, and making concerted efforts to understand and communicate to others how to put partial evaluation to practical use. Various "real-world" factors follow.

Real computer languages. This goes without saying; partial evaluators restricted to, say, Scheme or Haskell are unlikely ever to win widespread acceptance (alas...). Here recent progress in specialization of C and Fortran are quite relevant, and C++ is an obvious next step to take.

More convenient systems, not threatening or unconvincing. This category demands both smooth user interfaces and convincing demonstrations. It also requires that PE practitioners use *moderation* when stating the claims. If a speedup factor of 1000 is partly due to specialization and partly due to language and architecure changes, it can be hard to separate the individual threads entering into the result. People in, say, industrial practice, are more likely to be impressed by (and to believe) a speedup factor of two which is based on algorithms and languages which they use in daily practice.

Users should need to know their own program, and *not* `mix` itself. Again, this should go without saying; one can no more expect users to understand a perhaps continually evolving `mix` than one can expect him or her to understand the parse table compression techniques that Yacc uses.

Modularity in specialization. Most specializers require an entire program to be given in advance. Work on, for instance,modular binding-time analysis, is being done by Consel, Jouvelot and Ørbæk among others, but has not yet reached a maturity similar, for example, to that of ML's module system.

More predictable results of specialization, better termination properties. Again, some very desirable traits. Current work by P. H. Andersen and N. C. K. Holst holds promise for a good termination algorithm to add to Similix. Predicability of PE results will require, among other things, using the computer to estimate the speedup (if any) that PE can be expected to give. First steps in this direction have been taken by L. O. Andersen and C. Gomard.

Better control over code explosion. This was mentioned before; analyses should be devised to identify the possibility and cause of code explosion.

That PE specialists also look at *other people's programs*. There is a real danger of becoming rather inbred, as has been seen in other Computer Science communities. By its very nature PE has a broad spectrum of possible application areas and can help other people better solve their problems. Thus PE has both the possibility of and good motivation for wider external contacts, and these should be exploited.

5.3 Technology transfer

Once the technology matures enough to be used by "the man on the factory floor," there will be a clear need for special courses and other means of technology transfer. A first step in this direction is conference tutorials, and the two summer schools held at Carnegie-Mellon and in France.

6 On the role of open problems

The "state of the art" in many intellectual communities is very often expressed by a widely known collection of important *open problems* whose solution would advance that state. Such a collection, once developed under a broad consensus, can be an invaluable inspiration for further work and contribute significantly to the further maturing of the field.

In complexity theory "is P = NP?" and "does context free language recognition require superlinear time?" are central problems whose solution (positive or negative) would have widespread consequences. A more classical example: Hilbert's famous list of problems from the Mathematical Congress of 1900 greatly stimulated several branches of mathematics, including in particular recursive function theory and our understanding the fundamental limitations of formal mathematical systems.

6.1 Precision in Problem Definition and Criteria for Evaluating Solutions

One aim of this paper is to bring some of the many research directions in partial evaluation and mixed computation into sharper focus by listing a set of important problems, some near solution and some not. An essential point is to understand *just what it is that we aim to do*, in advance and in objective, communicable and evaluable terms. Essential criteria for selecting a problem as "challenging" include:

A precise definition What exactly is the problem to be solved?

Potential utility A problem to be solved should be useful and not just an intellectual curiosity: solutions should lead to insights that can be used on other problems, and lead to further investigations.

Abstraction It is vital that the principles used in a new program or system be described in terms as free as possible from the situational context in which it was created. Details to be abstracted away can include a particular programming language (with a special jargon, set of tricks, and ways of looking at problems); a particular user community; or a particular set of possible application areas.

If new results are presented abstractly there is a chance that *others can adapt the ideas* to novel applications, new languages, etc. If not, the only benefit others can gain is by copying and running the described programming system.

Judgement criteria Methods must be given if not apparent for ascertaining whether or not an alleged problem solution in fact is a correct solution.

6.2 Challenging problems

This section collects together a variety of problems in partial evaluation and mixed computation that appear to be worth solving but as yet lack solutions or even, in some cases, precise formulations. The first part concerns problems from the 1987 workshop on Partial Evaluation and Mixed Computation.

These and the following newer problems are not as definitive and far reaching as the known open problems from complexity and logic, and have a more temporary purpose: to encourage researchers in our field to clarify and understand their goals, to state them precisely, and thereby to move towards a broader and deeper understanding (needed to prepare more definitive future lists).

6.3 Challenging problems from PEMC 1987

Control and data. **3.1**: Can the residual program have essential loops or recursions that were not present in the source program? **3.2**: Can the source program have loops not present in the target program? **3.3**: Can suspension of parts of composite, structured data be achieved? Can PE be adapted to cope with data which is partially dynamic and partially static? **3.5** Can arbitrary residual programs be produced?

Answers: yes.

3.8, 3.9 on Strength and Nontriviality: *discussed above.*

3.4 Can the residual program have composite values that were not composite in the source? Can PE be used to generate new specialized data types, in a way analogous to generating specialized functions?

Answer: This has not yet been fully realized, though work is being done on *constructor specialization* by Torben Mogensen and Dirk Dussart.

Target code. **3.6**: Can mix-produced compilers and target programs achieve the efficiency of traditional runtime architectures? These architectures are efficient, well-developed and well-tested; and they seem at some abstract level both necessary and minimal. Can a traditional runtime architecture be derived automatically from the interpreter text? For example, can residual programs using techniques resembling Pascal's stack of activation records be automatically generated from an interpreter?

Answer: some progress has been made, for example by John Hannan.

3.7 Can traditional compiler techniques such as symbol tables, multiple passes, constant folding, etc. be derived automatically from `mix` and an interpreter by partial evaluation?

4.1 PE techniques typically yield target programs written in the interpreter's language. Can PE yield efficient low level machine code?

4.2 Can this be done by writing a metacircular interpreter, whose text uses only a machine-like subset of its source language? (The rationale is that the target program is a residual form of the interpreter and so inherits many of its characteristics.)

Answers: 4.1: N. C. K. Holst's AMIX does this, and Berlin and Weise generate parallel target code. 4.2: not yet achieved.

5.1 What is the source of the often large gains in efficiency yielded by even a very straightforward partial evaluation? **5.2** Can one automatically achieve more than linear speedups?

Answer: discussed above.

Automatic Tools for Program Generation

Partial evaluation raises the possibility of automatically transforming wider classes of nonprocedural problem specifications into algorithmic solutions. One application is parser generation. More generally, suppose there is a computable function $solve(problem\_specification, parameters)$ which outputs solutions when given acceptable problem specifications from a well-defined class, each with certain parameters. Then

```
finder = [[mix]](solve_program, probl_specification)
```

is a solution finder that given a parameter value yields a solution (grammar example: finder is a specialized parser mapping input strings into parse trees). Further

```
transformer = [[mix]](mix, solve_program)
```

transforms problem specifications into solution finders (e.g. transforms a general parser into a parser generator).

7.1 How far can this approach be carried in practice? Some examples:

problem specification	**solution finder**	**transformer**
context free grammar	parser	parser generator
interpreter	target program	compiler
set of Horn clauses	Prolog system	Prolog compiler
Categorical lang. def.	metainterpreter	interpreter
first order logic formula	specialized prover	prover generator
second order λ-calculus	*all*	*not*
constructive type theory	*nearing*	*yet*
higher order logic	*executability*	*automated*

Answers: Dybkjær, Mossin, and Thiemann have produced parser generators by PE. Many have generated compilers. Hans Dybkjær has realized Hagino's categorical framework for defining programming languages with good results.

Program Composition and Decomposition

Partial evaluation decomposes a program P into an executable part and an ejectable part. The first can be executed during partial evaluation, while the second is the residual program. (For interpreters, the executable part consists of all compile time actions and the ejectable part is the target program.) Each part has its own semantics and so there should be syntactic and semantic program composition operators $\oplus$, $\odot$ such that

P	=	executable part $\oplus$ ejectable part
semantics(P)	=	semantics(executable part) $\odot$ semantics(ejectable part)

Other program composition operators have been well studied, for example serial composition (homomorphisms, derivors, attribute coupled grammars) and parallel composition as well (the basis for divide and conquer algorithms). The problem is thus:

8.1 Study program composition and decomposition operators suitable for use in describing and implementing partial evaluation.

A Common Technical Problem: Generalization

The following problem appears to be the root of the termination problems that have plagued nearly all work in partial evaluation. Most partial evaluators work by constructing a set of configurations, each of which represents a set of run time computational states (or parts of them) that are reachable in computations on run time input data

satisfying the given input restrictions. For instance a configuration could contain the values of some but not all subject program variables. The known values we can call *static* and the others, *dynamic*.

There is a difficult balance to achieve, since configurations must be enumerated in enough detail to exploit what is known about the input, for thereby as much computation as possible can be done at PE time; but if too much detail is recorded there is a strong chance that the configuration set will be infinite and so lead to a PE-time "infinite loop". Call one configuration a *generalization* of another if it represents a larger set of run time computational states. The problem is thus to develop a good generalization strategy, marking enough of each new configuration as dynamic to guarantee finiteness but keeping enough static to get an efficient residual program.

9.1 Devise good strategies for generalizing configurations.

Answers: online and offline strategies have been briefly discussed above, but the problem is far from fully solved. The great majority of efficient fully self-applicable partial evaluators have used off-line generalization.

More Expressive Languages Much progress has occurred in this area.

Successful partial evaluators; see discussions above. **10.1** Functional languages with higher order functions. **10.3** Compilation of logic programming by PE. **10.4** Imperative languages with recursive procedures, references to nonlocal variables, structured data with pointers: C.

10.7 A self-applicable partial evaluator for term rewriting systems. *Done* by Bondorf.

10.8 Aid the efficient implementation of equational specifications (e.g . to aid the Knuth-Bendix algorithm). Some results by Sherman and Strandh.

Problems that are still not fully solved. 10.6 PE of machine, assembly or other low-level language **10.2** Languages with lazy evaluation. **10.5** Object-oriented languages. **10.9** PE of nondeterministic and/or parallel languages Can the semantics be preserved? What does this mean?

Specific Problems with Broader Implications Some progress has been made in these areas, but decisive results are still lacking.

11.1. Extend existing languages, e.g. with automatic insertion of debugging code, profiling or other instrumentation.

Type checking. **11.2** Achieve type security and yet do as few run time type tests as possible. (Being worked on by Dussart, Henglein, and Welinder.)

11.3 Determine from an interpreter whether the language it interprets is strongly typed; and exploit this information to generate highly efficient residual programs.

11.4 Derive from an interpreter an algorithm to do static type checking on the interpreted programs (as much as possible, and by automatic methods).

Program transformations and metacompiling. **11.5** Apply partial evaluation to a self-interpreter in order to generate an automatic program transformer, for example:

- from recursive form to tail recursive form;
- from lazy evaluation to eager evaluation
- by automatically introducing memo functions

11.6 Implement a truly user-extensible language with acceptable efficiency.

11.7 Semantics directed compiler generation by PE.

Answer. done by Jørgensen (see above).

More Distant Horizons **12.1**. Use partial evaluation to make practical use of the "Second Recursion Theorem" from recursive function theory (Kleene, Ju. L. Ershov).

12.2. Develop a system which will observe program runs and adaptively specialize a program when it is seen to be spending much time on a restricted class of input data. Applications:

- automatically compiling a frequently run program;
- automatically generating a compiler from a frequently run interpreter;
- implementing closures in a high-level language by partial evaluation

12.4. Automatically construct a program to estimate a subject program's running time.

12.5. Apply the latter to a "divide and conquer" algorithm to equalize the sizes of subproblems for distribution over a network of parallel processors.

12.6 Study relationships between learning and partial evaluation.

6.4 Some new challenging problems

In spite of the enormous scope of the preceding list, a few new problems have arisen within the past decade. Here are a few; the reader can doubtless supply more.

Dependence on specific architectures and compilers. PE sometimes interacts in unexpected ways with specific architectures and compilers. One problem is that complers are "tuned" to handwritten code, and machine-generated code often has very different characteristics. Another is that newer computer architectural features such as pipelines, and data and instruction caches at two or more levels can make it hard to predict whether, for example, unrolling a loop with static bounds will speed up execution. In principle it could even slow it down, due to increased cache misses, though I haven't yet seen solid evidence of this problem.

Symbolic operations on programs. Many operations on programs can be imagined on program texts that realize well-known mathematical and other operations on the meanings of those texts: composition, realizing , currying, application, tupling, inversion, etc. The problem, of course, is to realize them *efficiently*.

One answer to this is Phil Wadler's "deforestation," a form of symbolic composition, which has neen studied further by Chin, Sørensen, and is partly being implemented in the Glasgow Haskell compiler.

Combining the best features of deforestation and specialization. Deforestation can do some program optimizations beyond the reach of traditional PE. A system combining the best features of both would be quite interesting.

Self-applying and fully automating Turchin's supercompilation. Turchin's supercompilation can do both deforestation and specialization. A more fully automated version of supercompilation, based on a more traditional programming language than Refal, would be quite interesting.

Metasystem transition Turchin's work is based on the philosophical concept of *metasystem transition*, and this led him to realize what could be accomplished with self-application of a PE, already in the 1970s. It appears that this framework may have more consequences within programming language processing. One is the following topic.

Multiple levels of specialization. Why just two binding times? For some applications multiple specialization levels are natural, but few formulations and experiments have been made. Open questions include how to express "bias" as mentioned, and what mixed level analogues of the Futamura projections would be.

Glück and Jørgensen have done experiments realizing *specializer projections* that generalize the Futamura projections, and are currently doing multilevel experiments.

A better definition of "optimality." *A weakness in the definition of an "optimal" PE.* Unfortunately there is a fly in the ointment. The condition proposed above is expressed *relative to one particular self-interpreter* `sint`. It could therefore be 'cheated', by letting `mix` have the following structure:

```
read Program, S;
if Program = 'sint then
Result := S else
Result := the trivial specialization of Program
to S;
write Result
```

On the other hand, it would be too much to demand that `mix` yield optimal specializations of *all possible* self-interpreters. Conclusion: the concept of 'optimality' is pragmatically a good one, but one which mathematically speaking is unsatisfactory. This problem has not been resolved at the time of writing, and so could be a research topic for a reader of this paper.

Specialization to adapt to changing contexts Examples here can include generating special purpose code tailor-made to various parallel computing environments; and automatically producing highly efficient kernel code for operating systems, cryptography, and protocol implementation.

7 Self-critique

It is sometimes hard to see from published works on partial evaluation just *what problem has been solved*, let alone how the solution can be extended or adapted to other languages, program transformations or application areas. In spite of rapid progress and impressive new results in PE, there is still a large gap between the state of the art of our current research and its potential significance.

Many parts of Computer Science, including our own, lack an international intellectual community analogous to those in atomic physics, mathematical logic, numerical analysis or complexity theory (just to name a few). These communities have well established fundamental concepts, agreement as what the to major problem areas are, and the ability and tradition to find the essential contribution of a new piece of work, seeing beyond small methodological and terminological differences. We must do what we can to bring our community closer to this state.

7.1 Towards Greater Scientific Maturity

An important problem: there is all too little research in the classical meaning of the term, with the result that many works "reinvent the wheel" and omit highly relevant references to others' work. Another problem: very few papers discuss at all the limitations of their results and methods, so the reader is left to puzzle over which part of that which was not said can be routinely filled in, and which part is completely beyond the reach of the methods presented.

A relevant example: there is widespread parallel development of partial evaluation ideas within the functional progamming and the logic programming communities; yet far too little discussion across these groups' boundaries.

These problems hinder forming a true scientific community. It would be far more helpful for each new scientific contribution to emphasize the *commonality* that usually exists between it and its predecessors rather than superficial differences, to use a *consistent notation and terminology* (innovating only where essential), and to clarify *both the powers and limitations* of the new results.

Acknowledgements Special thanks are due to Robert Glück and Dirk Dussart for discussions on this paper, and to the Topps group at DIKU for more things than I can mention. Finally, I would like to acknowledge the many people whose research has contributed to this field. The following bibliography contains only a few of those mentioned in this paper, but a full one would be too extensive for these proceedings. The curious are invited to look first at [20], then at the bibiography of the Topps group at DIKU on the web, and if these fail, to send e-mail to me personally to ask for references.

A Personal Chronology in PE

1971, the Futamura projections:

- Discovered by Yoshihiki Futamura, published in an obscure Japanese journal
- Moral: self-application can generate programs that run faster!

Middle 1970s:

- Self-application independently rediscovered by Turchin, Ershov in the USSR, and

- Sandewall's group in Sweden (Uppsala, Linköping)
- Lectures in the West by Ershov on the potential of partial evaluation

1977-1979, a dream: Semantics-directed compiler generation

1980: A conference *Semantics-Directed Compiler Generation*, Aarhus 1980

Idea of binding-time separation (in a denotational semantics): Dave Schmidt, NJ

1981, 1982:

- The CERES system (Christiansen, NJ)
  - Generated a true compiler generator
  - Complex and not very strong
- Partial evaluation of Prolog at Linköping: Komorowski
- (or perhaps 1982): I first hear about self-application of a partial evaluator, at Linköping

1983: Meeting with Andrei P. Ershov

- I first understand the full potential of partial evaluation
- Meet Ershov at Paris IFIP 1983, ask "Does mix exist?" Answer: "not yet"
- Initial fascination with self-application, and focus for long after

1984: Breakthrough

- Bulyonkov's article on polyvariant specialization
- DIKU, in November: the first running nontrivial self-applicable mix
  - First-order statically scoped Lisp
  - Hand annotations for call unfolding
  - Typically:
    - target 10 times faster than running the interpreter
    - compiler 3 times larger than the interpreter

1985-1986: Reflection, experiments

- Simple **binding-time analysis** finding data dependency: mark as "dynamic" all expressions that depend on inputs not available to mix
- First conference paper on self-applicable PE, at *Rewriting Techniques and Applications*, Nancy
- Just what was it, really, that we had accomplished?
- Why did it work?
- Formalize and better understand binding-time analysis
- Applications:
  - Toy interpreters
  - Parser generation: Dybkjær
  - Ray tracing in a functional language: Mogensen
- First fully automatic mix: Sestoft
- Unmix system in Moscow: Romanenko
- Conference: *Programs as Data Objects*
- Valentin Turchin as guest professor at Copenhagen

1987: PEMC (Partial Evaluation and Mixed Computation) conference in Denmark (70 people)

- *Expensive* North-Holland proceedings
- Double issue of *New Generation Computing*
- Mix with *partially static data structures*: Mogensen
- Projections for specialisation: Launchbury
- Mix with *machine code as output language*: Holst
- *Automatic call unfolding*: Sestoft

1988:

- Partial evaluation bibliography, anonymous FTP: Sestoft
- Mix for programs in form of *term rewriting systems*: Bondorf
- Visit by NJ to researchers Romanenko, Klimov in Moscow, Bulyonkov in Akademgorodok
- Death of Ershov

1989: Some breakthroughs

- First journal paper on self-applicable PE: *Lisp and Symbolic Comutation*, Nancy
- Mix for a simple imperative language: Gomard, NJ
  - mix and a mix-generated compiler, in full detail
- Mix for *higher-order functional languages*: two
  - Lambda-mix for the *lambda calculus with constants, fix*: Gomard, NJ
  - Similix for *the Scheme language*: Bondorf
- Invited talk, Logic From Computer Science: NJ
- French Ph.D. thesis on partial evaluation of Scheme: Consel
- A prize-winning Ph.D. thesis on PE in a typed language: Launchbury (Glasgow)

1990: Progress, many experiments

- Book on the CERES compiler generator: Tofte
- Implement Hagino's categorical programming language framework: Dybkjær
- Invited talk, ICALP: NJ
- PE in scientific computing (IEEE Computer): Berlin and Weise

1991: Progress, American recognition

- Yale conference PEPM = *Partial Evaluation and Semantics-Based Program Manipulation*: 100 people, SIGPLAN Notices
- Tutorials and invited talks:
  - PEPM tutorial at Yale: Launchbury

  - POPL tutorial at Orlando, FPCA tutorial at Boston, AMAST (Algebraic Methodology and Applications to Software Methodology: NJ
- Better understanding: the types of compilers, interpreters, partial evaluators
- Compiling a sizable lazy language *from a denotational semantics* into Scheme: Jørgensen
- Partial equivalence relations for BTA: Hunt, Sands
- PE of an object-oriented language: Steensgaard
- Release of Similix partial evaluator for Scheme: Bondorf, Jørgensen
  - Anonymous FTP, widely fetched
  - User manual
  - "Binding-time debugger"
- Binding-time analysis by efficient type inference: Henglein

1992: Progress, many experiments

- PEPM workshop: San Francisco
- A self-applicable PE for a C subset: L. O. Andersen
- A self-applicable PE for the pure λ-calculus: Mogensen

1993:

- Efficient BTA and other analyses for Similix: Bondorf, Jørgensen
- PEPM conference: Copenhagen
- POPL tutorial at Charlestown: Consel, Danvy
- Book (Prentice Hall) on partial evaluation: NJ, Gomard, Sestoft
- Logimix for a Prolog subset: Bondorf, Mogensen
- SML-mix for a subset of Standard ML: Birkedal, Welinder
- Special issue on PE of J. Functional Programming
- Special issue on PE of J. Logic Programming

1994:

- PEPM workshop: Orlando
- Self-applicable Gödel partial evaluator: Gurr (Bristol)
- The essence of driving and PE: NJ
- PE for ANSI strictly conforming C: L. O. Andersen
- PE for Fortran 77 subset: Vienna (Glück as supervisor)
- Specializer generation: Glück, Jørgensen

1995:

- PEPM conference: San Diego
- Book (MIT Press) on partial evaluation: Consel, Danvy

References

[1] Andersen L.O., Program analysis and specialization for the C programming language. *DIKU, Department of Computer Science, University of Copenhagen.* DIKU Report No. 94/19, 1994.

[2] Baier R., Glück R., Zöchling R., Partial evaluation of numerical programs in Fortran. In: ACM SIGPLAN *Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* 119-132, Report 94/9, University of Melbourne, Australia 1994.

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

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

[5] Birkedal L., Welinder, M., Partial Dvaluation of Standard ML. *DIKU, Department of Computer Science, University of Copenhagen.* DIKU Report No. 93/22, 1993.

[6] D. Bjørner, A.P. Ershov, and N.D. Jones (eds.), *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, Gammel Avernæs, Denmark, October 1987*, Amsterdam: North-Holland, 1988.

[7] A. Bondorf and O. Danvy, 'Automatic autoprojection of recursive equations with global variables and abstract data types', *Science of Computer Programming*, 16:151–195, 1991.

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

[9] A. De Niel, E. Bevers, and K. De Vlaminck, 'Partial evaluation of polymorphically typed functional languages: The representation problem', in M. Billaud *et al.* (eds.), *Analyse Statique en Programmation Équationnelle, Fonctionnelle, et Logique, Bordeaux, France, Octobre 1991 (Bigre, vol. 74)*, pp. 90–97, Rennes: IRISA, 1991.

[10] A. P. Ershov: Mixed Computation: Potential applications and problems for study. Theoretical Computer Science 18, pp. 41-67, 1982.

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

[12] Robert Glück and Andrei V. Klimov, Occam's razor in metacomputation: the notion of a perfect process tree. In *Static analysis Proceedings*, eds. P. Cousot, M. Falaschi, G. Filé, G. Rauzy. Lecture Notes in Computer Science 724, pp. 112-123, Springer-Verlag, 1993.

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

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

[15] C. Gurr, A Self-applicable Partial Evaluator for the Logic Programming Language Gödel, Ph.D. thesis, University of Bristol, 1994.

[16] N.D. Jones (ed.), *Semantics-Directed Compiler Generation, Aarhus, Denmark, January 1980 (Lecture Notes in Computer Science, vol. 94)*, Berlin: Springer-Verlag, 1980.

[17] N.D. Jones and D.A. Schmidt, 'Compiler generation from denotational semantics', in N.D. Jones (ed.), *Semantics-Directed Compiler Generation, Aarhus, Denmark (Lecture Notes in Computer Science, vol. 94)*, pp. 70–93, Berlin: Springer-Verlag, 1980.

[18] N.D. Jones, P. Sestoft, and H. Søndergaard, 'An experiment in partial evaluation: The generation of a compiler generator', in J.-P. Jouannaud (ed.), *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pp. 124–140, Berlin: Springer-Verlag, 1985.

[19] N.D. Jones, 'Automatic program specialization: A re-examination from basic principles', in D. Bjørner, A.P. Ershov, and N.D. Jones (eds.), *Partial Evaluation and Mixed Computation*, pp. 225–282, Amsterdam: North-Holland, 1988.

[20] Neil D. Jones, C.K. Gomard, P. Sestoft, "Partial Evaluation and Automatic Program Generation," Prentice Hall International Series in Computer Science, 1993.

[21] Jones, N. D., *Constant time factors do matter*, ACM Symposium on Theory of Computing, ACM Press, S. Homer (ed.) (1993), 602-611.

[22] N. D. Jones,*The Essence of Program Transformation by Partial Evaluation and Driving*, in *Logic, Language and Computation, a Festschrift in honor of Satoru Takasu,* edited by Masahiko Sato N. D. Jones, Masami Hagiya, pages 206–224, S-V, April 1994.

[23] J. Jørgensen, 'Generating a compiler for a lazy language by partial evaluation', in *Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992*, pp. 258–268, New York: ACM, 1992.

[24] J. Launchbury, *Projection Factorisations in Partial Evaluation*, Cambridge: Cambridge University Press, 1991.

[25] S.A. Romanenko, 'A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure', in D. Bjørner, A.P. Ershov, and N.D. Jones (eds.), *Partial Evaluation and Mixed Computation*, pp. 445–463, Amsterdam: North-Holland, 1988.

[26] D. Sahlin, 'The Mixtus approach to automatic partial evaluation of full Prolog', in S. Debray and M. Hermenegildo (eds.), *Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, October 1990*, pp. 377–398, Cambridge, MA: MIT Press, 1990.

[27] Morten Heine Sørensen, Robert Glück and Neil D. Jones, Towards unifying partial evaluation, deforestation, supercompilation, and GPC. *European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, Springer-Verlag, 1994.

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

[29] M. Tofte, *Compiler Generators. What They Can Do, What They Might Do, and What They Will Probably Never Do*, volume 19 of *EATCS Monographs on Theoretical Computer Science*, Berlin: Springer-Verlag, 1990. Earlier version: DIKU Report 84/8, DIKU, University of Copenhagen, Denmark, 1984.

[30] Philip L. Wadler, Deforestation: transforming programs to eliminate trees. European Symposium On Programming (ESOP). Lecture Notes in Computer Science 300, pp. 344-358, Nancy, France, Springer-Verlag, 1988.