The Genesis of Mix: Early Days of Self-Applicable Partial Evaluation (Invited Contribution)

Peter Sestoft
IT University of Copenhagen
Denmark
sestoft@itu.dk

Harald Søndergaard
University of Melbourne
Australia
harald@unimelb.edu.au

Abstract

Forty years ago development started on Mix, a partial evaluator designed specifically for the purpose of self-application. The effort, led by Neil D. Jones at the University of Copenhagen, eventually demonstrated that non-trivial compilers could be generated automatically by applying a partial evaluator to itself. The possibility, in theory, of such self-application had been known for more than a decade, but remained unrealized by the start of 1984. We describe the genesis of Mix, including the research environment, the challenges, and the main insights that led to success. We emphasize the critical role played by program annotation as a pre-processing step, later automated in the form of binding-time analysis.

CCS Concepts: • **Software and its engineering** → **Translator writing systems and compiler generators**; Functional languages; • **Theory of computation** → **Program analysis**; • **Social and professional topics** → *History of software*.

Keywords: Partial evaluation, mixed computation, Lisp, self-application, auto-projector, compilation, compiler generation

ACM Reference Format:
Peter Sestoft and Harald Søndergaard. 2024. The Genesis of Mix: Early Days of Self-Applicable Partial Evaluation (Invited Contribution). In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM '24), January 16, 2024, London, UK*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3635800.3637445

1 Introduction

The concept (though not the name) of *meta-programming* is as old as that of the stored-program computer, both being present in Turing's seminal paper [42] on computability. It is the ability to treat programs as data that enables a program to be designed to read, generate, analyse and/or transform other programs, or even to modify itself while running. It allows the creation of *interpreters*: programs that execute programs, such as the "universal computing machine" [42], or Universal Turing Machine as we now know it.

A special case of meta-programming is *self-application*: a program given itself as input. Examples include an editor being used to edit its own text, as well as a compiler used to compile itself (assuming the language it is written in is a subset of the language it translates). Or, as the focal points in this paper, an interpreter that interprets itself, or a program specializer that specializes itself.

In this paper we recount the birth, 40 years ago, of "Mix", the first partial evaluator successfully used for non-trivial compiler generation via self-application. We describe the path that, with its ups and downs, led us to a design that finally worked. A critical insight was the need for so-called binding-time analysis, and we explain its role in self-application. Along the way we describe the environment that we worked in, as it arguably helped us succeed.

In Section 2 we briefly sketch related work that preceded Mix, and we describe the development of Mix (including dead ends) in some detail. One aim of this paper is to demystify the role of binding-time analysis. To prepare the stage for that, Section 3 recapitulates the so-called Futamura projections. Section 4 then explains the role of binding-time analysis, and Section 5 sketches how it led to full automation of Mix. In Sections 6 and 7 we discuss the legacy of the late Neil D. Jones and the crucial role he played in numerous programming technology projects. Section 8 summarizes and concludes.

While we try to make the paper reasonably self-contained, many technical details are necessarily left out. In keeping with an informal style, we shall often use "Neil" for Neil D. Jones, and "Peter" and "Harald" for the authors of this paper.

2 The History of Mix

2.1 Self-Application and Partial Evaluation

At first it might be feared that self-application begets paradoxes. For example, consider this program (call it M):

> When given input P:
> (1) Run P with input P
> (2) If the result is 0, output 1; else output 0

Running M on itself appears to create a contradiction, as the result of that application must be 0 if and only if it is 1. But

PEPM '24, January 16, 2024, London, UK

ACM ISBN 979-8-4007-0487-1/24/01
https://doi.org/10.1145/3635800.3637445

that reasoning assumes that there *is* a result; it assumes step (1) is completed in finite time. Once possible non-termination is taken into account, any paradox evaporates.

Indeed, constructive self-application of programs can have great utility. It has a history that dates back at least to the early 1950s. Corrado Böhm, in his 1952 doctoral thesis [4], published in 1954, (a) defined an abstract machine, faithful to contemporary machines and shown by Böhm to be Turing complete,[1] (b) defined a "high-level" programming language somewhat prescient of FORTRAN, and (c) presented a compiler for this language, *written in the language itself*. Ten years later, a compiler for Lisp 1.5, written in that language, was built at MIT [21]. The particular case of self-application that is of interest in this paper is that of a partial evaluator, a scenario first considered in 1971 by Yoshihiko Futamura.

We first knew about Futamura's ideas through the writings of Andrei Ershov [13, 14] (though we soon found that they were present already in Beckman *et al.* [2]). This is not the place for a more detailed account of related work preceding the Mix project; the interested reader is referred to Jones, Gomard and Sestoft [23] and the extensive bibliography by Sestoft and Zamulin [39]. Suffice it to say that partial evaluation of Lisp programs had been advocated and implemented by Lombardi and Raphael as early as 1964 [28, 29], and that, in the 1970s, the main efforts to develop Lisp partial evaluators took place at Linköping University in Sweden [2, 11, 19, 20]. We briefly return to related work in Section 3 when we discuss the so-called Futamura projections.

2.2 DIKU: The Early Days

The Computer Science Department at the University of Copenhagen [43] is known as DIKU, short for "Datalogisk Institut ved Københavns Universitet". DIKU was founded in 1970, with Peter Naur as foundation professor.

While this paper's main focus is on technical details of the early development of Mix, we start by pointing to three facets of DIKU in the 1970s and 1980s, to place our research activities around partial evaluation in their proper socio-historical context.

First, Naur had created a distinctive curriculum for the 5-year MSc ("cand. scient.") degree in Computer Science. In addition to coursework units, every student had to complete several variable-sized software development or research projects, culminating in a sizeable masters thesis. Notably, each project, irrespective of type, had to be carefully documented in the form of a written "report", the structure of which had to follow fairly rigid guidelines. Moreover, student peer review was hardwired into the curriculum: For a student to complete the MSc degree, it was not sufficient to complete the necessary number of coursework units and produce the prescribed amount of written work. Written work underwent student peer reviewing, and the reviewing duties were mandatory. The peer reviews, with their summaries and critique, were usually important input for instructors grading student projects, but the main purpose was for students to practise professional skills: to read and summarize technical writing crisply, to analyse and evaluate, and to provide opinion and feedback constructively. Already in 1969, Naur had described his "project activity" approach and the philosophy behind it [33]. We — two DIKU alumni who were subjected to it, and who proceeded into academic careers — later came to see the approach as innovative and pedagogically well ahead of its time. The emphasis on project work, written communication, and peer review helped develop, we think, a scholarly culture among students at DIKU. It developed not only technical and communication skills, but also judgement and affective skills. In conjunction with the use of older students as teaching assistants, it also fostered collaboration among students at different year levels, as well as between students and academics.

Second, DIKU was a small, close-knit community where students enjoyed considerable influence on both administration and academic decision making.[2] A variety of social events were held regularly which involved both students and staff. As a result there was strong community feeling, a sense of belonging, at least for students who chose to participate in committees and social events.

Third, DIKU was housed in a refitted factory, rather than a building designed for office work, let alone academic work. And yet the physical space turned out to be surprisingly fit for purpose (it is still used by sections of a now much larger DIKU). There was no dedicated student common room, nor was there a staff room of the "staff only" type then commonly found in universities. Instead, students, academic and professional staff mingled in the one and only "canteen" — a volunteer-run cafeteria equipped with a semi-industrial kitchen, offering basic food and drinks at low prices. The

[1] While the logicians' insights about computational universality were familiar to Böhm in the early 1950s, they were yet to be appreciated by computing experts generally. As evidence, Martin Davis [8] points to the following statement made in 1956 by the Harvard computing pioneer Howard Aiken: "If it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence that I have ever encountered."

[2] The day-to-day running of the department was the responsibility of the "institutbestyrer" who also chaired the "institutråd" — the department's executive committee, with representatives for academic staff, professional staff, and students. In 1971 the committee elected, with large majority, Flemming Sejergaard Olsen as its Chair. What is remarkable about that is the fact that Sejergaard Olsen was a student (who went on to discharge his tasks as Chair very competently). The newspaper "Politiken" commented that "the conditions at DIKU are somewhat strange, compared to the prevailing university climate; the students and academics collaborate and have done so since the institute was created a few years ago. Indeed, Sejergaard Olsen was nominated [as Chair] by academics" [34]. While the statutes governing the university technically allowed for a student to serve as the department manager, this was the only example of its kind ever, and a change of law soon put an end to such democratic indulgence.

canteen was a venue also for student-staff interaction, both academic and social.

Neil arrived in 1982, taking over the third year undergraduate course (Dat-2) which had a focus on programming language technologies and computability. This course gave Neil exposure to dozens of students and teaching assistants interested in theoretical computer science and hungry for project work in that area.

2.3 Late 1983: A Plan Is Hatched

It was during an informal chat in the canteen, in November 1983, that Harald, a teaching assistant at the time, told Neil about a fascinating paper he had studied earlier that year. In Nils Andersen's course on program transformation, Harald had been given the task of reading and presenting Andrei Ershov's "On the essence of compilation" [13]. The paper showed how, in theory, a self-applicable partial evaluator (or "auto-projector") may be used for compilation. To say that this topic resonated with Neil is an understatement. Neil declared, with great excitement, that the paper was currently sitting on his desk, he liked it very much, and he knew its contents in and out. He had in fact met Ershov in Paris (possibly at the IFIP'83 congress) a few months earlier, and had learned that no one had yet succeeded in constructing a non-trivial self-applicable partial evaluator. Since that meeting Neil had put considerable thought into how an auto-projector might be realized and used for all sorts of interesting experiments.

The starting point for a design was to settle on a programming language to use. There was a strong tradition of doing partial evaluation in Lisp [2, 11, 20, 29], and Lisp had also been favoured by Futamura [15]. Neil was also convinced that Lisp was the right choice; however, it seemed necessary to identify a suitable subset of pure Lisp, selected specifically with self-application in mind. The language would have to be powerful enough to allow the writing of a clever symbolic-manipulation tool, but at the same time it should be as simple as possible, to minimize bookkeeping and the number of programming constructs the tool had to deal with. We suspected that previous projects' commitment to large languages had been the bane of self-application. In modern terminology, what was needed was a well-designed domain-specific language (DSL) for the task.

Neil suggested a meeting where he and Harald could work out the details, both of the DSL and of the partial evaluator. Work thus began on the Mix project, with several meetings in Neil's office in December 1983. Neil was very confident that the project would succeed, and soon! As he was going to attend POPL in Salt Lake City the following month, mid January 1984 seemed like a natural deadline for a working auto-projector. It would be great to be able to demonstrate it to colleagues at POPL, said Neil.

2.4 First Half of 1984: Frustration

The problem turned out to be harder than expected. In one sense, building a partial evaluator is straightforward. Kleene showed this in the context of partial recursive functions, in the "S-m-n theorem" [27]. Given a representation of a function f of $m + n$ variables, you can easily construct a representation of an n-ary function g which is f specialized to fixed values $k_1, \ldots, k_m$ for its first m variables. You simply define

$$g(y_1, \ldots, y_n) = f(k_1, \ldots, k_m, y_1, \ldots, y_n)$$

so as to "hardwire" the given inputs $k_1, \ldots k_m$ into the representation.

But a partial evaluator worth its name really should be doing a fair amount of symbolic evaluation. It should perform certain basic program transformations, including constant folding, that could improve the runtime performance, and it should do this across function calls. In fact, doing this and doing it well would be essential in the context of self-application.

In the first half of 1984, a sequence of more and more sophisticated (or baroque, perhaps) auto-projectors were produced, each referred to as Mix. Each rested on the assumption that there would be a way for an auto-projector to make sensible decisions, on-the-fly, about how to process a function call: either "unfold" it, that is, continue with symbolic evaluation of the function called, or else "suspend" it, that is, leave the call as a residual expression, part of the generated residual program. But each of our more and more complicated unfold-or-suspend decision procedures turned out to fail. When self-applied, Mix would either fail to terminate, or else fail to symbolically execute function calls far enough.

In early 1984 two other students of Neil's, Mads Rosendahl and Torben Mogensen, worked on a partial evaluator for Pascal, but did not achieve self-application. The choice of Pascal was probably motivated by its role as the primary teaching language at the time, also in compiler construction. However, problems of parsing, encoding of tree data structures by pointers, manual memory management, Pascal's type system, and similar factors presumably hindered progress and experimentation.

That Peter got involved in the Mix project was due to a chance encounter in DIKU's canteen. As a third-year CS undergraduate, Peter had followed Neil's Dat-2 lectures on functional programming, interpretation and compilation with great enthusiasm. Much of that course used LetLisp, a version of Lisp with (statically scoped) let-bindings, compiled down to standard Lisp by a preprocessor which Neil had written in LetLisp itself. One afternoon in early 1984 Peter found some older students in the canteen listening to Mads

```
exp  →  (C S-expression)
     |  (INP)
     |  (ARG)
     |  (X)
     |  (HEAD exp)
     |  (TAIL exp)
     |  (PAIR exp exp)
     |  (IF exp exp exp exp)
     |  (CALL exp)
     |  (LETX exp exp)
     |  (REC exp exp)
```

Figure 1. An early version of a language L designed for ease of self-application

Tofte[3] (who Peter knew as a teaching assistant from the year before) talking about an exciting idea that Neil was working on. This involved not just programs transforming other programs, but programs transforming themselves, promising spectacular results if achieved in practice. Knowing Neil from the lectures may have emboldened Peter to approach him about this project.

In any case, Peter somehow joined Neil's and Harald's work on constructing a self-applicable partial evaluator, probably no later than April 1984, though the precise timeline is unclear. The best design of the language L that Mix would both process, and be written in (so as to be self-applicable) was still far from clear.

Up to that point, the successive versions of L were extremely simple languages without named entities, so a program could have only one (unnamed) function with one (unnamed) parameter in scope at a time. Figure 1 shows the syntax of such a language [40]. `(INP)` provided access to input. `HEAD`, `TAIL` and `PAIR` were like Lisp's `car`, `cdr` and `cons`; `C` was a "constancy" operator, like Lisp's `QUOTE`, and `IF` was the conditional operator, with `(IF` e_1 e_2 e_3 e_4`)` corresponding to the Lisp expression `(cond ((equal` e_1 e_2`)` e_3`) (t` e_4`))`. The language was call-by-value. A recursive function would be defined using `(REC` e_1 e_2`)`, with e_2 being the body of the function and e_1 its initial argument. Definitions could be nested, and a call `(CALL` e`)` would then always be to the function defined by the nearest enclosing `REC`. That way, functions and formal parameters need not be named — `(ARG)` would simply refer to *the* actual parameter. Similarly, let bindings were possible, but only to a single name, referred to through `(X)`. Accordingly, an expression `(LETX` e_1 e_2`)` meant "let `X` be bound to (the result of evaluating) e_1 in e_2".

While the lack of parameter (and let-variable) multiplicity made programming in L tedious, it was not a real restriction, as multiple parameters were readily encoded using Lisp data structures. Nevertheless, syntactic sugar was soon added: a single quote could be used in place of `C`, a right-associative infix "`::`" could be used in place of `PAIR`, and let clauses with pattern bindings were added — all compiled away by a preprocessor. With that, a self-interpreter could be presented as in Figure 2 and used as a basis for building auto-projectors.

```
(REC ((HEAD (INP)) :: '(() ()))
 (LET (eps . (beta . (alpha . xi))) = (ARG)
     (op e1 e2 e3 e4)               =  eps
     v1   =    (CALL (e1 :: (TAIL (ARG))))
     v2   =    (CALL (e2 :: (TAIL (ARG))))
     v3   =    (CALL (e3 :: (TAIL (ARG))))
     v4   =    (CALL (e4 :: (TAIL (ARG))))
 IN
 (IF op 'C    e1
 (IF op 'INP  (HEAD (TAIL (INP)))
 (IF op 'ARG  alpha
 (IF op 'X    xi
 (IF op 'HEAD (HEAD v1)
 (IF op 'TAIL (TAIL v1)
 (IF op 'PAIR (v1 :: v2)
 (IF op 'IF   (IF v1 v2 v3 v4)
 (IF op 'CALL (CALL (beta :: beta ::  v1   :: xi))
 (IF op 'LETX (CALL ( e2  :: beta :: alpha :: v1))
 (IF op 'REC  (CALL ( e2  ::  e2  ::  v1   :: xi))
     'error
 ))))))))))))
)
```

Figure 2. A self-interpreter for L [40]

The basic symbolic evaluation steps that would be required were fairly well understood. For example, the processing of an expression `(HEAD` e`)` would consist of symbolic evaluation of e, to obtain $e^\star$, followed by rewriting of the original expression to

`(C` a`)`	if $e^\star$ = `(C` $a.b$`)`
e_1	if $e^\star$ = `(PAIR` e_1 e_2`)`
`(HEAD` $e^\star$`)`	otherwise

We were prepared for the fact that decisions about when to unfold function calls would be the harder part to get right, and that this presumably would require much programming and experimentation. As it turned out, frugality had been taken too far in the design of L. In particular, the inability to name functions made experimentation cumbersome, so that progress was very slow. Nevertheless, the self-interpreter became the basis for a working, albeit not terribly powerful, partial evaluator, with dynamic decisions about call unfolding [40] — an approach later referred to as "online" partial evaluation.

[3] Mads Tofte at the time worked on Neil's CERES compiler generator, generalizing it using another form of self-application. He later contributed to the Standard ML language and to compile-time memory management, then founded the IT University of Copenhagen and headed it 1999-2018.

2.5 Mid 1984: The TOUPE Series

At some point in the summer of 1984 we instead decided that L should be a first-order "flat-scope" subset of Lisp, in which a program could have any number of (named) functions, each with any number of (named) parameters; essentially first-order recursion equations, or combinators. This was an improvement on two fronts. First, it made programming more natural and hence experimentation much easier and faster. Second, by accident or by (Neil's) design, unlike the previous languages, this one enabled a strategy of polyvariant program-point specialization, in which the residual program would be a collection of (0, 1, or more) specialized versions of each of the original program's function definitions. Indeed, most partial evaluators subsequently developed at DIKU use some form of polyvariant program point specialization [23], independently devised by Bulyonkov [7], for suitable notions of "program point".

For a while we worked with this language and with dynamic control (during partial evaluation) of call unfolding, initially with great optimism. Since the start of the project we had moved away from UNIVAC 1108 Lisp to running Franz Lisp on DIKU's new VAX 11/750, and later VAX 11/785, something that had resulted in good performance gains. However, we continued to run into situations where the generated residual programs would be either trivial, or infinite (so specialization would not terminate), or very large, due to flawed dynamic unfolding strategies. Peter recalls a situation where he had started a self-application experiment writing the residual program to a file (as a Lisp S-expression), gone to lunch, and when coming back met an irate computer operator because the residual program had consumed all remaining disk space on the departmental server (some 20 MB). We now pessimistically referred to our successive partial evaluator designs as TOUPE, for Tomorrow's Outdated Useless Partial Evaluator.

2.6 Second Half of 1984: A Breakthrough

A breakthrough came when, early in the second half of 1984, we began development of a version of Mix which abandoned the idea of on-the-fly unfolding decisions. Instead it was assumed that some kind of pre-processing would annotate each function argument, to indicate whether it should be treated as residual or not. Accordingly, the DSL was extended to allow for such annotations, in fact to allow for the annotation of *every* type of operation. This allowed us to experiment with annotations to see how they affected the generated compilers, and especially the size of these compilers.

With that, "offline" partial evaluation was born. In October 1984 we had arrived at a Mix version which performed well. It did require annotation, by hand, of function calls that appeared in the program being specialized. (In particular, Mix itself had to be thus annotated.) Based on this, a preprocessor annotated the program completely, that is, each operation *op* would be turned into "*op*-r" or "*op*-e", according as it should be left residual during partial evaluation, or symbolically evaluated. This finally led to the successful generation of compilers and compiler generators with manageable sizes and structures that could be studied and understood.

```
prog   →  (fundef fundef . . . )
fundef  →  (fname (var . . . ) exp)
exp    →  (quote S-expression)
       |  var
       |  (car exp)
       |  (cdr exp)
       |  (cons exp exp)
       |  (atom exp)
       |  (equal exp exp)
       |  (if exp exp exp)
       |  (call fname exp . . . )
```

Figure 3. The language L used for Mix self-application by late 1984 [35, page 10]

Our work up to that point (end of 1984) was written up as a conference paper submission to the *First International Conference on Rewriting Techniques and Applications* (RTA) [24]. The TR version [25] was published in January 1985 and the RTA'85 paper in May 1985.[4] As future work, the paper suggested that it "should be investigated whether a generally useful call annotation algorithm is possible" and indeed, we soon developed an automated call annotation method, based on binding-time analysis; see Section 5.

2.7 A Language for Self-Applicable Partial Evaluation

As mentioned, the language L used for successful self-application of Mix was a subset of statically scoped pure Lisp, essentially corresponding to first-order recursion equations, described in [35, 36]. The language is shown in Figure 3, where the notation "*exp* ..." indicates a sequence of zero or more occurrences of the grammatical construct *exp*. The language keywords are in lowercase, to better distinguish them from the earlier versions of L.

A program *prog* is a list of one or more function definitions. A function definition is a three-element list of a function name *fname*, a list of zero or more function parameters *var*, and the function body which is an expression *exp*. An expression *exp* is essentially a pure Lisp expression, but there are only nine expression forms, and a function call has the explicit syntactic constructor `call`.

In retrospect, even the Figure 3 version of L was possibly too parsimonious for its own good. Adding a `let`-binding to the core L language would have permitted us to separate

[4] Although the paper was only marginally within the RTA scope, it is today, according to Springer, the most cited paper from the 1985 conference.

code duplication concerns from termination concerns when making decisions about call unfolding; see Section 5. Torben Mogensen clearly realized this in 1986, including `let` in core L in his MSc thesis [30, pages 10-11]. In a later publication [26], essentially the same language was called Mixwell, but its sugared version Mixwell$^+$ had additional features.

In any case, we preserved the idea already used in Section 2.4 to define a more convenient extended language LetL, with "syntactic sugar" constructs such as these:

- `let` and `where` data decomposition patterns; for instance, `let (op exp1 exp2) = exp in ...` would bind the three elements of list `exp` to variables `op`, `exp1`, and `exp2`
- `if-then-elsf-else` chains of conditionals
- a right-associative infix cons operator `::`
- logical operators such as `null`, `not`, `and`, and `or`
- a `list` operator for building lists

These LetL constructs were compiled to plain L by a simple pre-processor, as in Neil's Dat-2 course [22]. Using these extensions, a self-interpreter for L could be written as shown in Figure 4.[5]

3 The Futamura Projections

Before we discuss the important role played by binding-time analysis, let us recapitulate the applications to compiler generation that, for us, motivated self-application. This section's equations have been published countless times, but we reproduce these classics for contrast with more binding-time-explicit versions in Section 4.1 and 4.2, to help understand how non-trivial self-application succeeded. In the following, for simplicity, we will gloss over issues of termination and also the distinction between a program's text and its semantics (as an input-output function).

Let p be a program that takes two arguments d_1 and d_2. Ordinarily, the application of p to (d_1, d_2) would be evaluated to a result v in one stage:

$$v = p(d_1, d_2) \tag{1}$$

However, alternatively it may be evaluated in two stages, first using a partial evaluator *mix* to specialize p with respect to part of its input d_1, obtaining residual program r, then running the residual program on the remaining input d_2 to obtain the result v:

$$\begin{array}{lcl} r & = & mix(p, d_1) \\ v & = & r(d_2) \end{array} \tag{2}$$

More generally, a program *mix* is a *partial evaluator* if for any two-argument program p and input values d_1 and d_2,

```
((L-int (prog input) =
    (let (fname1 pars1 body1) . rest) = prog in
        (call Exp body1 pars1 input prog))
 (Exp (exp vars vals prog) =
    (let (op exp1 exp2 exp3)    = exp
        (call? fname . argexps) = exp in
    (if (atom exp) then
        (call Lookupv exp vars vals)
     elsf (equal op 'quote) then
        exp1
     elsf (equal op 'call) then
        (call Call  (call Lookupf fname prog)
                (call Pars argexps vars vals prog)
                    prog)
   else (let v1 = (call Exp exp1 vars vals prog) in
     (if   (equal op 'car)  then (car v1)
      elsf (equal op 'cdr)  then (cdr v1)
      elsf (equal op 'atom) then (atom v1)
      elsf (equal op 'if)   then
       (if v1 then (call Exp exp2 vars vals prog)
        else       (call Exp exp3 vars vals prog))
    else (let v2 = (call Exp exp2 vars vals prog) in
      (if   (equal op 'equal) then (equal v1 v2)
       elsf (equal op 'cons)  then (cons v1 v2)
       else (list 'SYNTAX 'ERROR: exp))))))))
 (Call (fundef vals vars) =
    (let (fname pars body) = fundef in
       (call Exp body pars vals prog)))
 (Pars (explist vars vals prog) =
    (let (exp1 . exprest) = explist in
    (if (null explist) then 'nil
     else (cons (Exp exp1 vars vals prog)
                (Pars exprest vars vals prog)))))
 (Lookupv (var vars vals) =
    (let (vn1 . vnr) = vars
         (vv1 . vvr) = vals in
   (if (null vars) then (list 'UNKNOWN 'VARIABLE)
     elsf (equal var vn1) then vv1
     else (call Lookupv var vnr vvr))))
 (Lookupf (fname prog) = ... similar to Lookupv ...)
)
```

Figure 4. A self-interpreter for L as of late 1984 [35, page 11]. Above, `L-int` is the main function; `prog` is the program to be evaluated; `vars` is a list of variable names; and `vals` a parallel list of their values.

the result of the two-stage evaluation equals that of the one-stage evaluation:

$$\forall p, d_1, d_2.\ \begin{array}{lcl} \text{if} & r & = mix(p, d_1) \\ \text{then} & r(d_2) & = p(d_1, d_2) \end{array} \tag{3}$$

The two-stage evaluation shown in (2) may be preferable over the one-stage evaluation in (1) when the first stage

[5] The programming of program transformers can give rise to whimsical bugs. One day we were amazed to find a generated residual program containing the novel message `'UNKNOWN 'ERROR:`, apparently cobbled together from the `'UNKNOWN 'VARIABLE` and `'SYNTAX 'ERROR:` messages found in the code of Figure 4.

performs a great deal of computation based on d_1, and the second stage needs to be performed for many different values of d_2.

A special case of this is when p is an interpreter *interp*, d_1 is a program *src* to be interpreted, and d_2 is that program's input d. Then the first computation stage corresponds to compilation of *src* to a target program *tgt*, and the second stage corresponds to running the compiled target program on its input d:

$$\begin{array}{lcl} tgt & = & mix(interp, src) \\ v & = & tgt(d) \end{array} \tag{4}$$

Since *mix* itself is a program that takes two arguments, one may specialize *mix* with respect to its first argument, here the interpreter *interp*. Then one obtains a compiler *comp*, which when given the second argument *src* will produce a target program *tgt*:

$$\begin{array}{lcl} comp & = & mix(mix, interp) \\ tgt & = & comp(src) \end{array} \tag{5}$$

In the first line above, the outermost *mix* is a running program which is applied to some representation *mix* of itself, such as its program text, or abstract syntax tree. As described in Section 1, such "self-application" is in no way paradoxical, but it is challenging to design *mix* so that the result of self-application is non-trivial.

Going one step further, one may specialize *mix* with respect to itself (rather than with respect to an interpreter), to obtain a compiler generator *cogen*. Subsequently applying *cogen* to an interpreter *interp* will produce a compiler *comp*:

$$\begin{array}{lcl} cogen & = & mix(mix, mix) \\ comp & = & cogen(interp) \end{array} \tag{6}$$

The equations (4) through (6) were called "Futamura's projections" by Ershov in a short 1980 paper in Japanese [12].

Finally, it follows from the definition of *mix* in (3) that applying *cogen* to *mix* produces *cogen* itself:

$$cogen = cogen(mix) \tag{7}$$

This equation shows that if we consider *mix* an interpreter-style specializer that takes a two-argument program and a value of its first argument, then *cogen* is the corresponding compiler-style specializer: a program that turns a two-argument one-stage program p into a generator *pgen* of specialized versions of p. In other words, *pgen* is a two-stage version of p, taking one argument in each computation stage:

$$\begin{array}{lcl} pgen & = & cogen(p) \\ r & = & pgen(d_1) \\ v & = & r(d_2) \end{array}$$

From this it should be clear that *provided* a program *mix* satisfying (3) exists, and *provided* it can produce non-trivial residual programs when specializing itself, one can compile programs, generate compilers, and even generate a compiler generator — that can reproduce itself! These were exciting prospects! And yet, they had not been realized in practice by early 1984.

Futamura's original 1971 paper [15] states only the first two projections, whereas the third appears in a 1973 internal report by Futamura, as described in a 1999 interview [16]. In a 1976 paper, Beckman *et al.* [2] mention Futamura's 1971 paper very briefly. They discuss self-application and the three projections, without linking any of that to Futamura's paper, possibly because they were interested in applications other than compiler generation. A 1977 report by Valentin Turchin [41] also expresses the three projections, without reference to Futamura, or Beckman *et al.* Thus it appears that some or all of the Futamura projections were rediscovered independently during the 1970s. None of those discoveries, however, led to successful self-application in practice. Beckman *et al.*, recognising the usefulness of a generator $cogen = mix(mix, mix)$ (in our notation) instead proceeded to construct it by hand.

4 Annotations and Binding-Time Analysis

Binding-time analysis is a static analysis over a two-valued abstract domain $\{D, S\}$. The analysis assigns, to each function variable v, one of the abstract values S ("static") or D ("dynamic"), according as the values taken by v depend only on given input, during partial evaluation. A D thus means the possible values of v could depend (also) on the input that has not been given[6]. Based on this information, every expression can then be classified as static or dynamic, namely, an expression is static if it is a variable assigned S, a constant, or an expression all of whose components are static. Unfolding decisions can then be made, pre-partial evaluation time, based on how expressions are annotated. In particular, the decision about whether a recursive function call should be left residual or not can rely on binding-time information. Section 5 describes a simple such decision procedure.

In general, such a static-analysis approach leads to less specialization performed by a partial evaluator, compared to a dynamic decision-making approach. However, self-application is a rather special case, in which layers of interpretation are piled up. In the following we show why, in that case, pre-determined static/dynamic information enables one to achieve *deeper* specialization.

4.1 A Partial Evaluator That Takes a Binding-Time Argument

It is instructive to revisit the formulas describing partial evaluation and self-application in Section 3, now taking binding-time analysis and annotated programs into account. This

[6]The early Mix work used the terms "known" and "unknown", but these were found to be rather anthropomorphic and therefore replaced by "static" and "dynamic". In a similar vein, the compiler generator *cogen* in Section 3 was initially called *cocom*, for compiler-compiler, but was renamed to avoid associations to COCOM, the *Coordinating Committee for Multilateral Export Controls* targeting East Block countries.

section and the next one are based on [37], which credits Anders Bondorf for inspiration.

Due to the use of binding-time analysis, the self-applicable Mix partial evaluator developed in 1984 could be more accurately described as taking three arguments: a program p, a binding-time description δ, and some static arguments d_1 for p. We call this version *mix3* to distinguish it from the *mix* in Section 3. For the special case of p taking two arguments, the first one static (S) and the second one dynamic (D), the revised version of the "mix equation" (3) becomes:

$$\begin{array}{lllll} \forall p, d_1, d_2. & \text{if} & r & = & mix3(p, SD, d_1) \\ & \text{then} & r(d_2) & = & p(d_1, d_2) \end{array} \qquad (3'a)$$

For the special case of p taking three arguments, the first two static (S) and the third dynamic (D), the revised "mix equation" is:

$$\begin{array}{lllll} \forall p, d_1, d_2, d_3. & \text{if} & r & = & mix3(p, SSD, \langle d_1, d_2 \rangle) \\ & \text{then} & r(d_3) & = & p(d_1, d_2, d_3) \end{array} \qquad (3'b)$$

Now that *mix3* itself takes three arguments, the Futamura projections (4) – (6) must be revised correspondingly. The compilation of a source program *src* by specialization of an interpreter *interp* looks like this, by (3′*a*):

$$\begin{array}{lcl} tgt & = & mix3(interp, SD, src) \\ v & = & tgt(d) \end{array} \qquad (4')$$

To specialize *mix3* with respect to the interpreter *interp*, we should provide also the binding-time description $\delta = SD$ as static input; only the source program *src* should be dynamic. Hence, to generate a compiler as in (5), we actually specialize *mix3* with respect to *interp* and *SD*, and so the binding-time description of the outermost application of *mix3* has to be $\delta = SSD$. Thus, by (3′*b*):

$$\begin{array}{lcl} comp & = & mix3(mix3, SSD, \langle interp, SD \rangle) \\ tgt & = & comp(src) \end{array} \qquad (5')$$

When going one step further, specialising *mix3* with respect to itself (rather than with respect to an interpreter), we actually specialize *mix3* with respect to the tuple $\langle mix3, SSD \rangle$ to obtain a compiler generator *cogen*3. This *cogen3* must be applied not just to the interpreter *interp* but to the tuple $\langle interp, SD \rangle$ which specifies that we want *cogen3* to produce a *comp* that expects a source program *src* but not the source program's input d, again by (3′*b*):

$$\begin{array}{lcl} cogen3 & = & mix3(mix3, SSD, \langle mix3, SSD \rangle) \\ comp & = & cogen3(\langle interp, SD \rangle) \end{array} \qquad (6')$$

Finally, applying this *cogen3* to *mix3*, in analogy with (7), we must actually apply *cogen3* not just to *mix3* but to the tuple $\langle mix3, SSD \rangle$, which specifies that we expect *cogen3* to produce a "compiler-style" specializer that expects two (static) inputs, namely, a program and a binding-time description, but not the program's partial inputs. By (3′*b*):

$$cogen3 = cogen3(\langle mix3, SSD \rangle) \qquad (7')$$

Now let us consider how the presence of binding-time information facilitates non-trivial self-application in (5′). The crucial point is that the binding-time description *SD* is given as a static argument to the second occurrence of *mix3*, the one to be specialized, together with *interp*. This specifies that the residual program *comp* must be one that inputs a source program *src* and produces a program that in turn inputs a data value d and runs *src* on d; in other words, *comp* is a compiler.

Conversely, if the binding-time description given together with *interp* had been *DS*, the residual program should be one that inputs a data value d and produces a program that in turn inputs a source program *src* and runs *src* on d; not our usual concept of a compiler.

Moreover, in case no binding-time description at all were given to the to-be-specialized *mix3* together with *interp*, the self-application in (5′) would have had to generate an overly general "compiler" that would be able to handle both of the above situations, as well as others. Clearly such a "compiler" would both be unlikely to be recognizable as one, and probably very complex.

Thus it is not the binding time description *SSD* given to the running *mix3* that matters in self-application. What matters is that the to-be-specialized *mix3 also* gets a binding-time description as a static argument, telling it which inputs will be available to the residual program and which ones will not.

Analogously, in (6′), it is the second occurrence of *SSD* that is important for successful self-application.

To summarize, the introduction of an explicit binding-time argument δ supports successful self-applicable partial evaluation because it provides a principled way to *pass binding-time information also to the to-be-specialized partial evaluator* in a self-application, as in (5′) and (6′). The next section shows how this was actually done in Mix, by annotating all operations in the programs to be specialized, based on a static binding-time analysis.

4.2 A Partial Evaluator Using Annotations

The three-argument partial evaluator *mix3* described in the previous subsection can be thought of equivalently as a two-argument partial evaluator that processes annotated programs. This is because an annotated program p_δ depends only on the program p and the binding-time description δ, and these are always given together in (5′) – (7′). Hence the *SSD* in equations (5′) and (6′): S for the to-be-specialized program, S for its binding-time description δ, and D for the program's input.

Let us call the partial evaluator that processes annotated programs *mixa*, to distinguish it from *mix* in Section 3 and *mix3* in Section 4.1, but really it is just an alternative presentation of *mix3*. Note that *mixa* takes two arguments: an annotated program p_δ, and the static arguments d of that program.

Restating Equations (4′) through (7′) in terms of *mixa* and annotated programs, we get the following equations.

The compilation of a source program *src* (not annotated) by specialization of an annotated interpreter $interp_{SD}$ looks like this:

$$\begin{array}{lcl} tgt & = & mixa(interp_{SD}, src) \\ v & = & tgt(d) \end{array} \qquad (4'')$$

When specializing *mixa* with respect to interpreter $interp_{SD}$ to generate a compiler as in (5), the interpreter's annotations now provide the binding-time information *SD* that was given explicitly in (5′); only the source program *src* is dynamic. Hence, the binding-time description of the outermost application of *mixa* has to be $\delta = SD$; that is, the *mixa* being specialized is annotated as $mixa_{SD}$. The interpreter is annotated as $interp_{SD}$ as in (4′′):

$$\begin{array}{lcl} comp & = & mixa(mixa_{SD}, interp_{SD}) \\ tgt & = & comp(src) \end{array} \qquad (5'')$$

When going one step further, specialising the specializer with respect to itself (rather than with respect to an interpreter), we actually specialize the annotated $mixa_{SD}$ with respect to the annotated $mixa_{SD}$ to obtain a compiler generator *cogena*. This *cogena* must be applied to an annotated interpreter $interp_{SD}$ specifying that we want *cogena* to produce a *comp* that expects a source program *src* (hence the *S*) but not the source program's input *d* (hence the *D*):

$$\begin{array}{lcl} cogena & = & mixa(mixa_{SD}, mixa_{SD}) \\ comp & = & cogena(interp_{SD}) \end{array} \qquad (6'')$$

Finally, applying this *cogena* to *mixa*, in analogy with (7′), we must actually apply *cogena* to the annotated $mixa_{SD}$, which specifies that we expect *cogena* to produce a "compiler-style" specializer that expects one (static) input, namely, an annotated program, but not the program's partial inputs:

$$cogena = cogena(mixa_{SD}) \qquad (7'')$$

Obviously, annotations support successful self-application for exactly the same reasons already given in Section 4.1; an annotated program p_δ simply has the binding-time description δ baked in.

The connection between *mixa* and *mix3* can be described in terms of a pre-processor program *annotate* that takes as input a program *p* and a binding-time description δ of *p*'s inputs, performs a binding-time analysis, and produces an annotated program p_δ. Then we could define

$$\begin{array}{lcll} mix3(p, \delta, d) & = & \text{let} & p_\delta = annotate(p, \delta) \\ & & \text{in} & mixa(p_\delta, d) \end{array} \qquad (8)$$

Thus the annotator *annotate* can be considered a preprocessor for the specializer proper, which is *mixa*. Now, isn't it cheating to apply the specializer *mixa* only to annotated programs p_δ, including annotated versions of itself, as in Equations (5′′) and (6′′)? No. If we apply it to the full *mix3*, then, since both *p* and δ are given (static), it would fully compute the p_δ term interpretively before specialising *mixa* in the next step — provided of course that the specializer *mixa* encompasses the functionality of an interpreter: when given all arguments, it can compute (a representation of) a value, rather than an expression that needs to be evaluated.

4.3 Problems Caused by Binding-Times

The preceding sections showed how binding-times and annotations facilitated successful self-application. However, their introduction also caused new problems, requiring new solutions.

Our rather simple binding-time analysis would treat a variable as either entirely static or entirely dynamic. Now consider an interpreter using the classical Lisp representation of an environment by a list of pairs `(var . val)`, each holding a variable's name and its value. If the interpreter were to be specialized with respect to a program but not the program's inputs, then the values would be classified as dynamic, and hence the entire environment including the variable names would be classified as dynamic. Consequently, the result of specializing the interpreter would typically be trivial.

The solution to this problem can be seen in the `Exp` function of the self-interpreter in Figure 4, where the environment is represented by parallel lists `vars` and `vals`, respectively holding all-static variable names and all-dynamic variable values. This replacement of a list of pairs by a pair of lists is an example of a "binding-time improvement", the rewriting of a program to compensate for the simplicity of the binding-time analysis. Later work proposed better binding-time analyses [31].

Another problem is illustrated by the `Lookupf` function in Figure 4, which finds the definition of function `fname` in the program `prog`. If this function were used in a partial evaluator that is being partially evaluated, then typically `prog` would be static but `fname` dynamic (coming from a list of function specializations yet to be computed), and so the result of `Lookupf` would be dynamic, causing the result of self-application to be trivial.

The solution to this problem is more subtle, exploiting "bounded static variation", for instance, the fact that a given program contains only finitely many function definitions. Essentially, instead of returning the function definition from `Lookupf` for processing elsewhere, the processing would be moved into the success case of the lookup loop. The resulting specialized program would then typically contain an if-else chain corresponding to the possible functions. This technique is known as The Trick [23, Section 5.4.3].

5 From Semi-Automation to Full Automation

The self-applicable partial evaluator we reported in May 1985 [24] performed automatic binding-time analysis (BTA) for variables and expressions as described in Section 4, and

automatically annotated all operations as eliminable or residual, as described in Section 2.6. However, function calls still had to be manually (and statically) classified as either `call`, meaning that the call will be unfolded, or `callr`, meaning that the call will be residual, that is, replaced by a call to a specialized function.

An automatic call unfolding strategy should strive to avoid generating trivial residual programs (by unfolding too little), avoid generating infinite residual programs (by unfolding too much), and avoid generating needlessly large or inefficient residual programs (by duplicating computations when unfolding). In the call unfolding strategies described below, an annotation stage would use the results of binding-time analysis to mark each function call as eliminable or residual, and then the subsequent function specialization stage would just obey these annotations without further cleverness, in keeping with the general philosophy of Mix.

A very simple strategy actually suffices [23, Section 4.4]:

1. If a call has only static arguments, then mark it eliminable.
2. Otherwise mark the call residual.

This approach will produce a very large number of very simple residual functions, and perhaps for this reason it wasn't seriously considered in early work. Instead, the first fully automatic call annotation strategy for Mix [26, 38], developed sometime during 1985–1986, was based on the concept of an inductive static variable, implemented by case 2 below:

1. If a call has only static arguments, then mark it eliminable.
2. Otherwise, if the call is a recursive call from a function `f(. . . ,vi,. . . )=. . .` to itself, and the call has a static argument position i that computes a substructure, for instance `(car vi)`, of that argument's previous value `vi`, and all other static variables are unchanged in the call, then mark it eliminable.
3. Otherwise mark the call residual.

In case 2 one must additionally ensure that no non-trivial argument in a position classified as dynamic might be duplicated by the unfolding, since that might lead to an explosion in the residual program's size (if such duplication happens recursively during partial evaluation) or lead to an explosion in the residual program's running time (if the duplicated residual expression contains a recursive call) [38]. In retrospect, had we added a `let`-binding to the core L language in Figure 3, then we would not have needed this extra caveat, nor the simple program analysis that implemented it. Moreover, the conceptual untangling of code duplication concerns from termination concerns might have allowed us to realize sooner that an automatic call annotation strategy was feasible.

At a later point we switched to a simpler fully automatic call annotation strategy, based on the notion of function call under static control [23, Section 5.5.1]:

1. If a call has only static arguments, then mark it eliminable.
2. Otherwise, if the call is not in a branch of a dynamic conditional `if`, then mark it eliminable.
3. Otherwise mark the call residual.

This is the approach used in the Scheme0 specializer [23, Appendix A]. The code duplication caveat above still applies to item 2.

With all of these three call unfolding strategies, the resulting specialized programs might contain many rather simple functions, some consisting of just a call to another function. Such near-trivial functions could then be removed by a postprocessing stage. The postprocessing stage analysed the residual program's call graph, decided on a cut-point (a function) in each simple call cycle, unfolded harmless residual function calls while avoiding duplication of code and computation, and also beautified the names of the generated residual functions [38, Section 4.2] [26, Section 6.5].

Putting together the pre-processing needed to turn a LetL program into an L program (Section 2.7), the binding-time analysis and annotation stage described by (8), and the postprocessing, it is clear that the specialization $r' = mix3(p, \delta, d)$ of LetL-program p with binding-time description δ and static input d, giving residual program r', works in these stages [26, Section 7]:

$$\begin{array}{lcll} q & = & desugar(p) & \text{from LetL to L} \\ q_\delta & = & annotate(q, \delta) & \text{BTA and annotation} \\ r & = & spec(q_\delta, d) & \text{function specialization} \\ r' & = & post(r) & \text{postprocessing} \end{array} \tag{9}$$

Hence, in the end self-application was achieved by careful design of the language `L` being processed, the invention of binding-time analysis, the invention of a static call unfolding strategy, and by separating the various processes and concerns into multiple stages.

6 The Legacy of Neil D. Jones

In retrospect, it is remarkable that almost all the early Mix work was done without any external funding, formal project structures, or the like. It is a testament to Neil's openness and generosity of mind that he allowed Peter, a random undergraduate student, to join his and Harald's work on solving a programming challenge that had been open for over a decade, despite efforts in Japan, Sweden and the USSR.

This spirit of openness and collaboration continued to characterize Neil's group as it grew considerably in the following years. Neil would acquire external funding, not least from the EU, for collaborative basic research projects, resulting in strong connections to Cambridge, Chalmers in Gothenburg, École Normale Supérieure in Paris, Edinburgh,

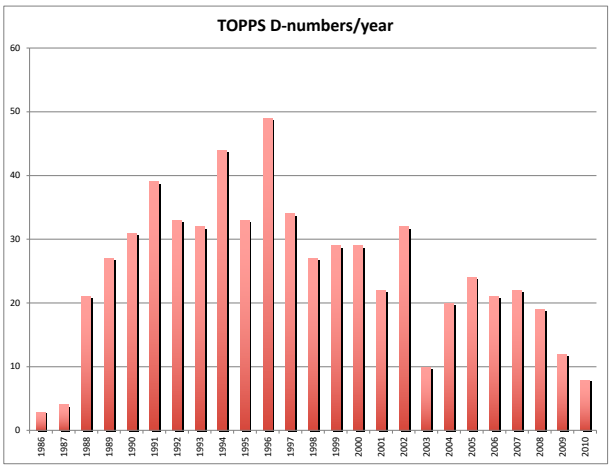

Figure 5. Number of TOPPS group publications per year (computed in 2014 from data on the group homepage)

Glasgow, Imperial College, many North American universities, Moscow, Novosibirsk, and other places. This in turn attracted many international postdocs, visitors and faculty and created a phenomenal scientific atmosphere in Neil's group, lasting for decades, and an international network of colleagues that still matters today.

From 1988, the group was named the TOPPS group, an acronym for the Danish equivalent of *Theory and practice of programming languages*. Figure 5 shows the vigorous TOPPS-related publication activity from around 1988, as a consequence of the international visibility and networking.

Neil was an outstanding researcher, educator and supervisor, showing great faith in students' abilities to solve problems, and liberally sharing advice, techniques and tricks when one got stuck. His 1984 lecture notes for the third year undergraduate course on programming languages and computability provided an admirable example of clarity and intellectual economy, covering a broad range of topics in just 103 type-written pages [22]. This approach influenced a generation of computer scientists, as expressed also in Neil's obituary [9].

The next section describes some of the work on partial evaluation at DIKU that followed the early Mix work. Among the many other subsequent contributions from Neil's group, it is worth mentioning the two very influential C pointer analyses that were created by his students Lars Ole Andersen (in 1994, motivated by partial evaluation of C) [23, Chapter 11] and Bjarne Steensgaard (in 1996, inspired by Fritz Henglein's work on efficient type inference). Both analyses have been cited more than 1500 times in the scientific literature. A variant of Steensgaard's method is part of the widely used LLVM compiler infrastructure today.

7 Further Work on Partial Evaluation

A large amount of ever more sophisticated work on partial evaluation followed after 1985, in Neil's group and elsewhere. Among the earliest are Hans Dybkjær's work on specialization of general parsers [10] and Torben Mogensen's work on specialization of ray tracers [30]. In other significant work, Anders Bondorf and Olivier Danvy developed Similix, a self-applicable partial evaluator for a higher-order subset of Scheme [5, 6], Carsten Gomard and Neil developed one for the lambda calculus [18], Lars Ole Andersen developed one for a substantial and useful subset of the C programming language [1], and Torben Mogensen and Anders Bondorf developed one for Prolog [32]. Many many later contributions followed.

By 1989, Neil proposed to collect all the then existing knowledge and work in a monograph, which appeared in 1993 [23]; thirty years later, it has been cited more than 2200 times. It was co-authored with Carsten Gomard[7], Peter, Lars Ole Andersen and Torben Mogensen. The book gives the full source code of a simple self-applicable partial evaluator for IEEE Scheme, close in spirit to the fully automatic Mix described in Section 5, apart from a simpler call unfolding strategy [23, Appendix A]. The book also describes partial evaluation developments and literature until early 1993 [23, Chapter 18].

Multiple international symposia on partial evaluation and related topics were co-organized by Neil or by people from his group. Neil and Harald Ganzinger organized an international workshop *Programs as Data Objects* at DIKU, October 1985, with proceedings [17]. An even more remarkable event was the week-long *Workshop on Partial Evaluation and Mixed Computation* in October 1987 at Gammel Avernæs in Denmark, which for the first time brought together Andrei Ershov, Neil D. Jones, Valentin Turchin, Yoshihiko Futamura, their current and former students, as well as many others, including notably John McCarthy, the inventor of Lisp. The workshop was organized and the proceedings edited by Dines Bjørner, Andrei Ershov and Neil D. Jones [3]. In 1991, a long-term home and venue for partial evaluation research was created by Olivier Danvy, who had been in Neil's group since 1986 and contributed considerably to it, and Charles Consel, in the form of the ACM symposium series *Partial Evaluation and Semantics-Based Program Manipulation* (ACM PEPM).

8 Conclusion

We have described the early history of self-applicable partial evaluation in the group of Neil D. Jones at DIKU, University of Copenhagen, and its historical and organizational

[7] Carsten Gomard was another of Neil's students, and at the time had developed a self-applicable partial evaluator for the lambda calculus. In 1999 Carsten co-founded Netcompany, now Denmark's largest software company by revenue.

context. We have pinpointed some technical aspects that, in retrospect, enabled non-trivial self-application, and we have briefly sketched how Neil's group blossomed in subsequent years.

Acknowledgements

We are grateful to the PEPM 2024 Program and History committees, not least Fritz Henglein, for the invitation to write this historical note. We also want to thank Mads Rosendahl, Mads Tofte, Olivier Danvy, Robert Glück and Torben Mogensen who provided invaluable help with early publications, recollections, references, and other matters. A great many other colleagues, friends, mentors and students contributed over the years; we regret that most must go unmentioned here.

References

[1] L. O. Andersen. 1993. Binding-Time Analysis and the Taming of C Pointers. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*. ACM Publ., 47–58. https://doi.org/10.1145/154630.154636

[2] Lennart Beckman, Anders Haraldson, Östen Oskarsson, and Erik Sandewal. 1976. A Partial Evaluator, and Its Use as a Programming Tool. *Artificial Intelligence* 7, 4 (1976), 319–357. https://doi.org/10.1016/0004-3702(76)90011-4

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

[4] Corrado Böhm. 1954. Calculatrices digitales. Du déchiffrage de formules logico-mathématiques par la machine même dans la conception du programme. *Annali di Matematica Pura ed Applicata* 37 (1954), 175–217. English translation at https://www.itu.dk/people/sestoft/boehmthesis/.

[5] A. Bondorf. 1991. Automatic Autoprojection of Higher Order Recursive Equations. *Science of Computer Programming* 17 (1991), 3–34. https://doi.org/10.1016/0167-6423(91)90035-v

[6] A. Bondorf and O. Danvy. 1991. Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types. *Science of Computer Programming* 16 (1991), 151–195. https://doi.org/10.1016/0167-6423(91)90002-f

[7] M. A. Bulyonkov. 1984. Polyvariant Mixed Computation for Analyzer Programs. *Acta Informatica* 21, 5 (1984), 473–484. https://doi.org/10.1007/bf00271642

[8] Martin Davis. 2012. *The Universal Computer: The Road from Leibniz to Turing*. CRS Press. https://doi.org/10.1201/b11441

[9] DIKU 2023. Obituary for professor emeritus Neil Jones. https://di.ku.dk/english/news/2023/obituary-for-professor-emeritus-at-diku-neil-jones/ 13 April 2023.

[10] Hans Dybkjær. 1985. Parsers and Partial Evaluation: An Experiment. DIKU Student Project Report 85-7-15.

[11] Pär Emanuelson and Anders Haraldsson. 1980. On Compiling Embedded Languages in Lisp. In *1980 Lisp Conf.* (Stanford, CA). ACM Publ., 208–215. https://doi.org/10.1145/800087.802808

[12] A. Ershov. 1980. About Futamura's Projections. *Bit (Japan)* 12, 14 (1980), 4–5. (In Japanese).

[13] A. P. Ershov. 1978. On the Essence of Compilation. In *Formal Description of Programming Concepts*, E. J. Neuhold (Ed.). North-Holland, 391–418.

[14] A. P. Ershov. 1982. Mixed Computation: Potential Applications and Problems for Study. *Theoretical Computer Science* 18 (1982), 41–67. https://doi.org/10.1016/0304-3975(82)90111-6

[15] Yoshihiko Futamura. 1971. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Systems, Computers, Controls* 2, 5 (1971), 45–50.

[16] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process, Revisited. *Higher-Order and Symbolic Computation* 12 (1999), 377–380. Question/answer style background information about Futamura's early work.

[17] Harald Ganzinger and Neil D. Jones (Eds.). 1986. *Programs as Data Objects. Copenhagen, Denmark, October 1985*. LNCS, Vol. 217. https://doi.org/10.1007/3-540-16446-4

[18] C. K. Gomard. 1992. A Self-Applicable Partial Evaluator for the Lambda Calculus: Correctness and Pragmatics. *ACM Transactions on Programming Languages and Systems* 14, 2 (April 1992), 147–172. https://doi.org/10.1145/128861.128864

[19] Anders Haraldsson. 1977. *A Program Manipulation System Based on Partial Evaluation*. Ph.D. Dissertation. Linköping University, Sweden. Linköping Studies in Science and Technology Dissertations 14.

[20] Anders Haraldsson. 1978. A Partial Evaluator, and Its Use for Compiling Iterative Statements in Lisp. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*. ACM, 195–202. https://doi.org/10.1145/512760.512781

[21] T. Hart and M. Levin. 1962. Memo 39—The New Compiler. MIT Computation Center memo.

[22] Neil D. Jones. 1984. *Datalogi 2 Notes: Functions, Expressions, Programming Languages, Computability*. Technical Report 84/7. DIKU, University of Copenhagen. https://di.ku.dk/forskning/Publikationer/tekniske_rapporter/1980-1984/Jones_Dat2Notes_1984.pdf

[23] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall. https://www.itu.dk/people/sestoft/pebook/

[24] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. 1985. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. In *Rewriting Techniques and Applications*, J.-P. Jouannaud (Ed.). LNCS, Vol. 202. Springer-Verlag, 124–140. https://doi.org/10.1007/3-540-15976-2_6

[25] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. 1985. *An Experiment in Partial Evaluation: The Generation of a Compiler Generator*. Technical Report 85/1. DIKU, The University of Copenhagen.

[26] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. 1989. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation* 2, 1 (1989), 9–50. https://doi.org/10.1007/BF01806312

[27] S. C. Kleene. 1938. On Notation for Ordinal Numbers. *Journal of Symbolic Logic* 3, 4 (dec 1938), 150–155. https://doi.org/10.2307/2267778

[28] L. A. Lombardi. 1967. Incremental Computation. In *Advances in Computers*, F. L. Alt and M. Rubinoff (Eds.). Vol. 8. Academic Press, 247–333.

[29] L. A. Lombardi and B. Raphael. 1964. Lisp as the Language for an Incremental Computer. In *The Programming Language Lisp: Its Operation and Applications*, E. C. Berkeley and D. G. Bobrow (Eds.). MIT Press, 204–219.

[30] Torben Mogensen. 1986. *The Application of Partial Evaluation to Ray-Tracing*. Master's thesis. DIKU, University of Copenhagen, Denmark.

[31] Torben Mogensen. 1988. Partially Static Structures in a Self-Applicable Partial Evaluator. In *Partial Evaluation and Mixed Computation*, D. Bjørner, A. P. Ershov, and N. D. Jones (Eds.). North-Holland, 325–347.

[32] T. Mogensen and A. Bondorf. 1993. Logimix: A Self-Applicable Partial Evaluator for Prolog. In *Logic Program Synthesis and Transformation: Proceedings of LOPSTR 92*, K.-K. Lau and T. Clement (Eds.). Springer-Verlag, 214–227. https://doi.org/10.1007/978-1-4471-3560-9_15

[33] Peter Naur. 1970. *Project Activity in Computer Science Education*. Consiglio Nazionale delle Richerche, I. E. I., Pisa, Italy. 13 pages.

[34] Politiken. 1971. Institutbestyrer studerer. 19 October 1971 issue, page 18.

[35] Peter Sestoft. 1985. *The Structure of a Self-Applicable Partial Evaluator*. Technical Report 85/11. DIKU, The University of Copenhagen. https://di.ku.dk/forskning/Publikationer/tekniske_rapporter/1985-1989/Sestoft-DIKU-report-85-11.pdf

[36] Peter Sestoft. 1986. The Structure of a Self-Applicable Partial Evaluator. In *Programs as Data Objects*, H. Ganzinger and N. D. Jones (Eds.). LNCS, Vol. 217. Springer-Verlag, 236–256. https://doi.org/10.1007/3-540-16446-4_14

[37] Peter Sestoft. 1987. Mix Takes Three Arguments. (1987). Handwritten note, 6 pages, 4 November 1987.

[38] Peter Sestoft. 1988. Automatic Call Unfolding in a Partial Evaluator. In *Partial Evaluation and Mixed Computation*, D. Bjørner, A. P. Ershov, and N. D. Jones (Eds.). North-Holland, 485–506.

[39] Peter Sestoft and Alexander V. Zamulin. 1988. Annotated Bibliography on Partial Evaluation and Mixed Computation. *New Generation Computing* 6, 2, 3 (1988), 309–354. https://doi.org/10.1007/bf03037145 Bibtex file at https://www.itu.dk/people/sestoft/pebook/partial-eval.bib.

[40] Harald Søndergaard. 1984. A Primitive Autoprojector for a Simple Applicative Language. DIKU Student Project Report 84-2-4, 83 pages.

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

[42] A. M. Turing. 1936. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 42 (1936), 230–265. https://doi.org/10.1112/plms/s2-42.1.230

[43] UCPH [n.d.]. UCPH Department of Computer Science. https://en.wikipedia.org/wiki/UCPH_Department_of_Computer_Science Accessed 17 November 2023.

Received 2023-10-20; accepted 2023-11-20