Extended Abstract

An Experiment in Partial Evaluation: The Generation of a Compiler Generator

Neil D. Jones Peter Sestoft Harald Søndergaard

Institute of Datalogy, University of Copenhagen
Sigurdsgade 41, DK-2200 Copenhagen N, Denmark

1. Introduction

The present paper is an extended abstract of (Jones 1985), in which a running, non-trivial partial evaluator is described. As far as we know, this partial evaluator is the first which has been used in practice to transform interpreters into corresponding stand-alone compilers, or to generate a compiler generator by transforming itself.

A partial evaluator is a program (call it mix) written in a programming language L, which takes as input a program p and a known value d_1 of p's first input argument. It produces as output a so-called residual program:

$$\boxed{\text{resid} = \text{L mix} \langle p, d_1 \rangle}$$

which, if run on p's remaining input $\langle d_2, \ldots, d_n \rangle$, will yield the same result as if p were run on all of its input data. (Notation: L ℓ $\langle d_1, \ldots, d_n \rangle$ denotes the output (if any) obtained by running L-program ℓ on input data $d_1, \ldots, d_n$).

Let int be an interpreter written in L which implements another programming language S. In (Futamura 1971) it was pointed out that partial evaluation could be used to compile from S into L:

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

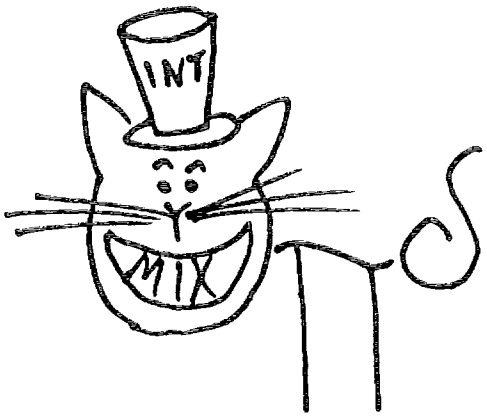
(where s is the S source program) and even to generate a compiler:

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

by partially evaluating the partial evaluator itself. The logical next step is the generation of a compiler generator by:

$$\boxed{\text{cocom} = \text{L mix} \langle \text{mix}, \text{mix} \rangle}$$

To our knowledge these promising constructions had not been carried out in practice prior to summer 1984, although some work in this direction has been done (see the list of references below).

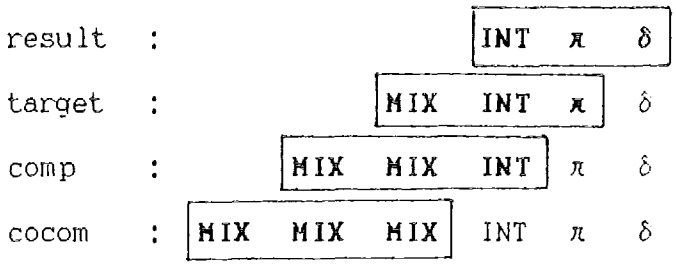


π : program

δ : data

'All right," said the cat; and this time it vanished quite slowly, beginning with the end of the tail, and ending with the grin, which remained some time after the rest of it had gone.

Lewis Carroll,
Alice's Adventures in Wonderland



2. Method

The choice of language L turns out to be crucial. The present L is a dialect of pure Lisp (with static scopes and without funargs), and a program is a system of functions defined by mutual recursion. A residual program consists of a set of variants of the original equations, each specialized to certain known argument values.

Partial evaluation is done in two stages. First, the program is flow analyzed (preprocessed) by a simple abstract interpretation, yielding as output an annotated version of the same program. Second, the annotated version is transformed into a residual program by applying a number of rewriting rules.

A critical problem is the processing of a function call - whether to *suspend* it (i.e. produce a residual call, possibly to a specialized function version), or to *unfold* it. The present partial evaluator refrains from taking such decisions. Instead, the user is required to classify calls as *residual* or *eliminable*.

The full paper describes both preprocessing and transformation in more detail. It also contains an analysis of some rather subtle problems which had to be solved before a running partial evaluator could be constructed. These include developing an adequate strategy for processing function calls, and analyzing cause and effect in the computation of L mix ⟨mix,int⟩, in particular as regards the size and speed of the resulting compiler.

3. Results

The compilers produced by mix have a surprisingly natural structure and are reasonably small and efficient, as the figures below indicate. Even cocom is of reasonable size, although its logic is harder to follow. It contains some unexpected constructions (like ''''nil !), and could be compacted by 10-20% by some simple peephole optimizations.

A mix-produced compiler works like a traditional compiler in that it translates a composite program structure by first compiling its substructures and then combining the resulting target program fragments. Unlike traditional compilers, the mix-produced ones first examine the fragments to see whether they contain operations that can be done at compile time, thus automatically doing what is often a separate pass: peephole optimization and constant folding. For example, the compiler for a simple flow chart language compiles the program X:=X+1; Y:=Y+1; X:=X-1 into the same code as Y:=Y+1.

Following are some results regarding size and run time of some of the target programs, compilers, and the compiler generator generated by mix. In the figures, "int" denotes an interpreter for a simple language S with list-valued expressions, while, if, and assignment, and "source" is an example S-program. Mix, target, comp, and cocom were produced as described above.

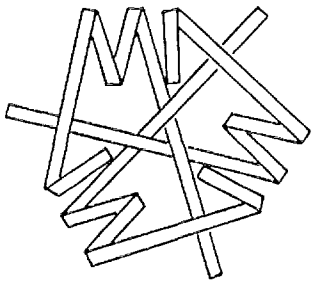
Program	# functions	# lines	Ratio
source	-	30	
target	13	53	1.8
int	9	130	
comp	39	474	3.6
mix	24	500	
cocom	126	1755	3.5

Figure 1: Size measurements

Run		Time in seconds			Ratio
output	= L int <source,data>	6.22	+	1.14	
	= L target data	0.46	+	0.56	7.2
target	= L mix <int,source>	7.20	+	1.78	
	= L comp source	0.28	+	0.00	32.1
comp	= L mix <mix,int>	433.80	+	37.20	
	= L cocom int	12.60	+	12.80	18.5
cocom	= L mix <mix,mix>	2617	+	203	
	= L cocom mix	74.80	+	55.50	21.6

Figure 2: Time measurements

The running times in Figure 2 are given in the form: computation time + garbage collection time. They do not include preprocessing, which is fast relative to partial evaluation. Note that comp and cocom run about 20 times as fast as their non-partially evaluated counterparts int and mix. For a more realistic interpreter, the first and second ratios would have been higher, and inclusion of preprocessing would make the last three higher.



4. Conclusions

The mix project was undertaken after a conversation with A. P. Ershov in Fall 1983, in which it was learned that no one had yet succeeded in constructing a non-trivial L mix ⟨mix,mix⟩ by computer. The problem turned out to be more subtle than expected, and we have had to throw away numerous earlier versions of mix for various languages. In particular, it was very difficult to anticipate the structure, size, or efficiency of a mix-produced compiler, although these have (at last, and somewhat to our surprise) turned out to be quite reasonable.

There were essentially three conceptual breakthroughs that were necessary to carry out the project. The first was the realization that interpreters *almost* always handle their structural argument by recursive descent, with the consequence that *calls* should be annotated as residual or eliminable (and not functions, as has been done with other partial evaluators). The second insight was to see the value of preprocessing the program, to classify each function's arguments as residual or eliminable *in advance*, rather than doing this adaptively *during partial evaluation*. The third came from an analysis of the effects of mix's structure and call annotations on the size of

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

and resulted in extending the preprocessor to classify each of an L-program's operations as residual or eliminable.

A drawback of the present solution is the need for annotating all function calls. This can of course be reduced by default assumptions (e.g. "all calls to f are eliminable unless otherwise specified"). A certain amount of automatic call annotation can certainly be done (since it is easy in Lisp to see when a known argument is replaced by a substructure), but it is not yet clear whether a generally useful and efficient annotation algorithm is possible. The full paper discusses objectives and pitfalls in connection with call annotation.

A somewhat different programming style is needed in order to realize the full benefits possible with partial evaluation. For one example, for compiler generation purposes the traditional interpreter's "A-list"

$$((\text{name}_1.\text{value}_1) \ldots (\text{name}_n.\text{value}_n))$$

is best split into a separate name list and value list. The reason is that the name list and its associated lookup and updating functions need not appear in the residual target program, since they depend only on the source program's syntax, whereas the value list *must* be present.

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

1. Programs can both accept programs as input data and produce them as output.
2. The language's semantics makes it easy to do symbolic evaluation and to do the flow analysis used by the preprocessor.
3. The natural recursivity of partial evaluation is easy to program.
4. The referential transparency of L programs facilitates specialization of an arbitrary program part without disturbing other parts. Although specializations constitute a limited class of transformations, they may imply considerable changes in program topology.

It would be more difficult to write a partial evaluator for an imperative language, due to the need for more sophisticated partial evaluation algorithms, and due to the difficulty of recognizing "descents" by a program into a smaller part of a structured known argument. On the other hand, logic programming languages such as Prolog seem to have all the desirable characteristics mentioned above, so it seems likely that a Prolog-based mix could be constructed.

5. References

Beckman, L. *et al*,
A partial evaluator and its use as a programming tool, *Art. Int.* 7,4 (1976) 319-357

Emanuelson, P. & A. Haraldsson,
On compiling embedded languages in Lisp, *Proc. 1980 ACM Lisp Conference*, Stanford, California (1980) 208-215

Ershov, A.,
On the partial evaluation principle, *Inf. Proc. Letters* 6,2 (April 1977) 38-41

Ershov, A.,
Mixed computation: potential applications and problems for study, *Theor. Comp. Sci.* 18 (1982) 41-67

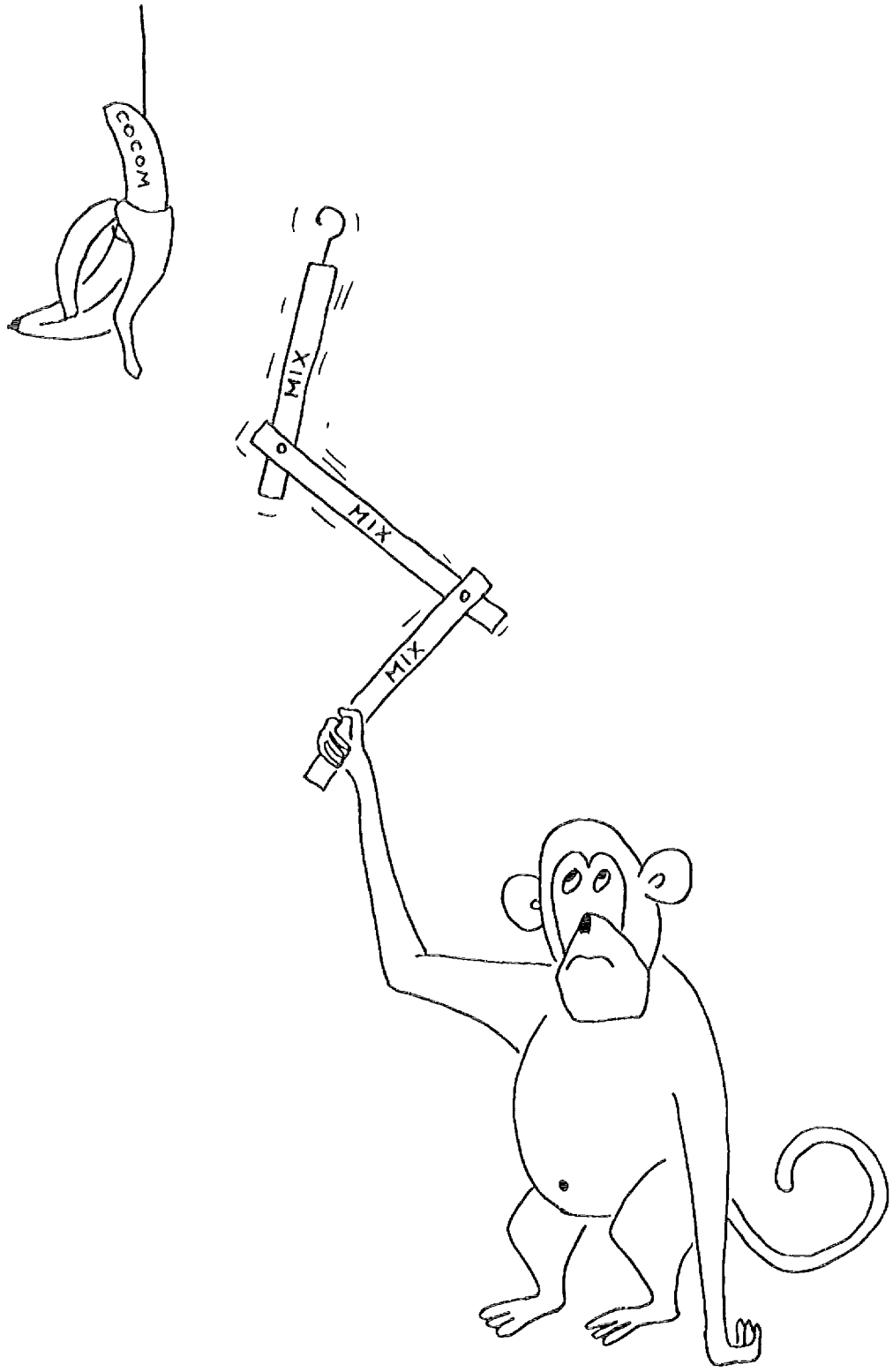
Futamura, Y.,
Partial evaluation of computation process - an approach to a compiler-compiler, *Systems, Computers, Controls* 2,5 (1971) 721-728

Jones, N. D., P. Sestoft & H. Søndergaard,
An Experiment in Partial Evaluation: The Generation of a Compiler Generator, DIKU Report no. 85/1, University of Copenhagen, Denmark (1985)

Kahn, K. & M. Carlsson,
The compilation of Prolog programs without the use of a Prolog compiler, *Proc. Int. Conf. Fifth Generation Computer Systems*, Tokyo, Japan (1984) 348-355

Turchin, V.,
Semantic definitions in REFAL and the automatic production of compilers, in *LNCS 94: Semantics-Directed Compiler Generation* (N. D. Jones, ed.) 441-474, Springer 1980

Turchin, V. *et al*,
Experiments with a supercompiler, *Proc. 1982 ACM Symp. on Lisp and Functional Programming*, Pittsburgh, Pennsylvania (1982) 47-55


COCOM
MIX
MIX
MIX