Extended Abstract of

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

Neil D. Jones
Peter Sestoft
Harald Søndergaard‡

DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark

INTRODUCTION: Since the early seventies it has been known that in theory, the program transformation principle called partial evaluation can be used for compiling and compiler generation, and even for the automatic generation of a compiler generator. A successful implementation of an experimental partial evaluator able to generate stand-alone compilers and compiler generators, however, had not been carried out prior to 1984 when the first mix system was brought to work at the University of Copenhagen.

In this paper we discuss partial evaluation and its applications to compiler generation. We also sketch the current mix system (mid 1987). The results we report are sufficiently remarkable to justify further research into using partial evaluation for compiler generation purposes.

A partial evaluator may be thought of as a "smart interpreter". If an interpreter is given a program and only part of this program's input data, it will leave the program unevaluated and report an error. A partial evaluator will attempt to evaluate the given program as far as possible.

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

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

Furthermore, partially evaluating a partial evaluator with respect to a fixed interpreter yields a compiler for the language implemented by the interpreter. And partially evaluating the partial evaluator with respect to itself yields a compiler generator, namely, a program that transforms interpreters into compilers.

In the rest of the paper we will make this a little more formal, give an example of partial evaluation, and sketch the structure of the partial evaluator mix.

†The full version of this paper is to appear in the journal LISP and Symbolic Computation
‡Current address: Computer Science Dept., Univ. of Melbourne, Parkville Vic. 3052, Australia
1. PRELIMINARIES

In this section a framework is set up for discussing partial evaluation and its applications. Our definition of a programming language may seem a bit pedantic at first sight. A precise notation is necessary, however, since more than one language may be discussed at the same time, and programs can play multiple roles: sometimes as active agents, sometimes as passive data, and even sometimes as both at once.

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

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

We identify a programming language $L$ with its semantic function on whole programs (assumed to be computable):

$$L: D \rightarrow D \rightarrow D.$$ 

The well-formed $L$-programs are those to which $L$ assigns a meaning:

$$L\text{-programs} = \text{domain } L.$$ 

The input-output function computed by $l \in L$-programs is $(L \ l): D \rightarrow D$ (which is partial since $l$ may loop). Thus $L \ l <d_1, ..., d_n>$ denotes the output (if any) obtained by running the $L$-program $l$ on input data $<d_1, ..., d_n>$. For an example, consider the following program power to compute $x$ to the $n$'th power:

$$\begin{align*}
\text{power} = & \quad f(n, x) = \begin{cases} 
1 & \text{if } n = 0 \\
\text{else if } \text{even}(n) \text{ then } f(n/2, x)^2 \\
\text{else } x \times f(n-1, x)
\end{cases}
\end{align*}$$

The result of running the program power is the result of applying its first function $f$ (the goal function) to the program's input values. For example, $L \text{ power } <3, 2> = 8$. 