AN EXPERIMENT IN PARTIAL EVALUATION:
THE GENERATION OF A COMPILER GENERATOR

Neil D. Jones
Peter Sestoft
Harald Søndergaard
DIKU
University of Copenhagen
Sigurdsgade 41, DK-2200 Copenhagen N., Denmark

1. Introduction

It has been known for several years that in theory the program transformation principle called partial evaluation or mixed computation can be used for compiling and compiler generation (given an interpreter for the language to be implemented), and even for the generation of a compiler generator.

The present paper describes an experimental partial evaluator able to generate stand-alone compilers and compiler generators. As far as we know, such generations had not been done in practice prior to summer 1984.

Partial evaluation of a subject program with respect to 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 can be considered 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 the program's parameters.

1.1. Application to Compiler Generation

The significance of partial evaluators for compiling, compiler generation, and compiler generator generation stems from the following fact: Consider an interpreter for a given 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. From this viewpoint then, partial evaluation and compilation are nothing but special cases of program transformation for the purpose of optimization.

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

It is nearly always easier to implement a new language by writing an interpreter than by writing a compiler for the language since in the latter case, one has to think of two binding times, compile time and run time. Interpretive implementations have only one binding time, but are usually too

'Presented May 1985 at the First International Conference on Rewriting Techniques and Applications, Dijon, France
inefficient for practical use. The potential significance of a good partial evaluator is thus that it allows for the automatic construction of efficient compilers from more intelligible interpretive specifications of programming languages (by automatically splitting the binding time into two).

In our experience, the compilers produced turn out to be natural in structure, reasonably efficient, and to produce reasonably efficient target programs, which typically run one order of magnitude faster than the interpreted source programs.

It could be argued that the restriction to language definitions in interpretive form is too limiting, since in practice one often chooses to define languages by denotational or axiomatic semantics, rather than operationally. However, denotational semantics provides "runnable specifications" of programming languages (as shown by the existence of several semantics-based compiler generators: [Mos79], [Set81], [Pau82], [ChJ83]), and these may in principle be regarded as interpreters.

1.2. Relation to Other Work

A very extensive list of references to partial evaluation literature is given in [Fut83]. The applicability of partial evaluation to compiler generation is mentioned in [Fut71], and to compiler generator generation in [Tur80], and [Ers82]. Some steps have been taken towards realizing these goals in practice ([Bec76], [EmH80], and [Tur82]), but as far as we know, we are the first to give a complete solution to this open problem.

Our partial evaluator differs from those presented in the literature mainly by being itself written in the same language it partially evaluates, and thus being self-applicable (whence it is called an autoprojector). This is essential for the generation of compilers and compiler generators.

Also, it uses a separate stage of preprocessing to determine statically (by flow analysis) which parts of a program may be reduced away during partial evaluation, and which parts will possibly appear in the residual program produced. The partial evaluators reported in the literature determine this dynamically during partial evaluation, but preprocessing appears to be advantageous for the realization of compiler generation and compiler generator generation. Finally, while this preprocessing is done automatically, the user is required to annotate (by hand, presently) the function calls appearing in the subject program.

1.3. The Paper

First, this paper gives an introduction to partial evaluation and presents the results from its application to compiler generation. Second, the subject language and the autoprojector are described, and special attention is paid to some rather subtle problems which had to be solved during the construction. Finally, the project is reviewed, and suggestions for further work are given.