Partial Evaluation of C and Automatic Compiler Generation

(Extended Abstract)

Lars Ole Andersen

DIKU, Department of University of Copenhagen,
Universitetsparken 1, DK 2100 Copenhagen Ø, Denmark.
E-mail: lars@diku.dk

Abstract. A partial evaluator is a program transformator which as input take a program and parts of its input, and as output produce a specialized residual program. When applied to the rest of the input data, the residual program yields same result as the orignal program. The aim is efficiency: the residual program often runs an order of magnitude faster. We have developed a self-applicable partial evaluator for a substantial subset of the C programming language. The possibility of self-application enables generation of stand-alone compilers from executable specifications, for example interpreters.

1 Introduction

During the last decade *partial evaluation* has proven its usefulness in numerous areas ranging from specialization of scientific computation [3], to specialization of general scanners [9], to automatic compiler generation [4,5]. The aim of partial evaluation is efficiency. Until recently, only languages with a clean semantics have been subject for partial evaluation, but as the techniques are better understood, more pragmatically languages can be considered, *e.g.* the C programming language.

Usually, interpreters are easier to write than compilers. For example, it is a manageable task to program a simulator for a machine architecture but a similar compiler may require a considerably effort. Compilers are here to stay, however, for one indisputable reason: efficiency. A compiled version of a program often runs an order of magnitude faster than an equivalent interpreted version. Partial evaluation can bridge the two worlds, giving both the advantages of prototyping via general interpreters, and the efficiency of compilers. By specialization of an interpreter to a program, the net effect is compilation, and if the partial evaluator is self-applicable, an interpreter can automatically be transformed into a compiler.

Similar holds in general. Programs taking two inputs where the latter varies more often than the former, usually benefits from specialization with respect to the first input. For example, a general scanner is a program which takes as input a token specification and a stream of characters. By specialization with respect to the token specification, a fast lexer tailored to the particular set of tokens is produced.

We have developed a self-applicable partial evaluator for a substantial subset of the widespread C programming language. The aim of the paper is to report the work done, technical details can be found in [2,1].

2 Partial Evaluation and Programs as Data Objects

A partial evaluator is a program operating on programs as data objects. It is therefore important to distinguish between a program text, its meaning, and a representation. Let p be the text of a program. By $[\![p]\!]_C(d_1, \ldots, d_n) \Rightarrow d$ we denote the result d of applying a $\mathcal{C}$ program p to the data d_i. Suppose t is the type of an unspecified data structure. By $\overline{p}^t$ we denote the *representation* of p as a t data structure. The notation is overloaded to also describe the representation of an arbitrary data in a particular data structure.

In a typed language, types, value representations and self-application interferes. Let `sint` be a self-interpreter for C. In languages like Scheme, programs are legal data objects and can be given directly to a self-interpreter, but in C, a suitable program representation must be devised. Furthermore, as the self-interpreter is supposed to work on programs taking different types of input data, the input data must be *encoded* into a single uniform data type `Val`. In C, this will typical be a `struct` with a huge `union` inside. Applying the self-interpreter to a program p and input data d_i corresponds thus to $[\![\texttt{sint}]\!]_C(\overline{p}^{\texttt{Pgm}}, \overline{d_i}^{\texttt{Val}}) \Rightarrow \overline{d}^{\texttt{Val}}$, where `Pgm` is a data structure for programs. Suppose the self-interpreter is self-applied: run on itself with input data p and d_i. As input to the interpreted version of `sint`, p must be represented as a `Pgm` data structure, but as input to the running version of `sint`, it must subsequently be encoded into the `Val` type. That is:

$$[\![\texttt{sint}]\!]_C(\overline{\texttt{sint}}^{\texttt{Pgm}}, \overline{\overline{p}^{\texttt{Pgm}}}^{\texttt{Val}}, \overline{\overline{d_i}^{\texttt{Val}}}^{\texttt{Val}}) \Rightarrow \overline{d}^{\texttt{Val}},$$

where the program and input data has been *double encoded*. One of the problems connected with the double encoding is the memory usage. This can be alleviated by "tricks" [6].

Recall the operation of a partial evaluator `mix` as is stated in the Mix Equation.

Definition 1 (The Mix Equation). Let p be a program, and d_s, d_d input to it. A partial evaluator `mix` is a program such that $[\![\texttt{mix}]\!]_C(\overline{p}^{\texttt{Pgm}}, \overline{d_s}^{\texttt{Val}}) \Rightarrow \overline{p'}^{\texttt{Pgm}}$ and $[\![p']\!]_C(d_d) \Rightarrow d$ iff $[\![p]\!]_C(d_s, d_d) \Rightarrow d$ (provided execution of `mix` terminate).

The input d_s given to `mix` is called the *static input*, the rest, d_d, the *dynamic input*. The `mix` produced program p' is called a *residual program* and is a specialized version of p with respect to d_s. The possibility of compilation, stand-alone compiler generation and even compiler generator generation, is contained in the Futamura projections, restated below with explicit value and program representations.

Proposition 2 (The Futamura Projections). *Let* `mix` *be a partial evaluator and* `int` *an (unspecified) interpreter.*

1. *Futamura* $[\![\texttt{mix}]\!]_C(\overline{\texttt{int}}^{\texttt{Pgm}}, \overline{\overline{p}^{\texttt{Pgm}}}^{\texttt{Val}}) \Rightarrow \overline{\texttt{target}}^{\texttt{Pgm}}$ *Compilation*
2. *Futamura* $[\![\texttt{mix}]\!]_C(\overline{\texttt{mix}}^{\texttt{Pgm}}, \overline{\texttt{int}}^{\texttt{Pgm}}{}^{\texttt{Val}}) \Rightarrow \overline{\texttt{compiler}}^{\texttt{Pgm}}$ *Compiler*
3. *Futamura* $[\![\texttt{mix}]\!]_C(\overline{\texttt{mix}}^{\texttt{Pgm}}, \overline{\texttt{mix}}^{\texttt{Pgm}}{}^{\texttt{Val}}) \Rightarrow \overline{\texttt{cogen}}^{\texttt{Pgm}}$ *Compiler Generator*

The first projection shows that compilation can be done by specialization of an interpreter to a program. The second, involving self-application, demonstrates stand-alone compiler generation, and the third, the generation of a compiler generator. Proofs of the projections can be found in *e.g.* [5].

3 Partial Evaluation of C

The basic principles in partial evaluation is *specialization* of program points to known values, *reduction* of partially known expressions, and *evalution* of known expressions. Hence, every statement (expression) must be classified as being either specialization-time or run-time. This can either be done *during* the specialization (on-line) or *before* the specialization (off-line) (by the means of a binding time analysis). On-line techniques have been advocated for imperative languages [7], but when self-application is desired, experience clearly shows that the off-line method is more appropriate. Furthermore, using off-line techniques, information about effects may be given to the specializer enabling better control of these. We solely consider off-line specialization, but not binding time analysis. In [1] an automatic binding time analysis based on type inference and constraint set solving is given.

The treated subset of C is transformed into an intermediate language Core C, similar to "three address" code. This provide an explicit control-flow, and furthermore, side-effects are isolated (by the means of assignment and call statements).

The supported data structures include base type variables (`int`, `char` *etc*), structures, multi-dimensional arrays and pointers to arrays. General pointers to and heap allocated structures is not handled. In[1], specialization of dynamically allocated arrays is described, and handling of heap allocated structures is discussed. A formal definition of Core C in form of an operational semantics can also be found.

3.1 Specialization of Statements

For explicit separation of the binding times, a *two-level* Core C language is exploited. Underlined constructs are dynamic (run-time) and non-underlined constructs are static (specialization-time). The specialization can then be formalized as a *two-level* semantics over the two-level language, cf. [1].

Example 1. Consider the from partial evaluation canonical `power` program computing x to the nth. Suppose `n` is static (3) but `x` dynamic. A two-level Core C version is shown below to the left.

```
int power (int n int x)
{  int pow
   1: assign pow = 1
   2: if (n) 3 6
   3:    assign pow = pow * x
   4:    assign n = n - 1
   5: goto 2
   6: return pow
}
```

```
int power_3(int x)
{  int pow
   1: assign pow = 1
   2: assign pow = pow * x
   3: assign pow = pow * x
   4: assign pow = pow * x
   5: return pow
 }
```

At specialization time, the specializer executes the non-underlined statements, and generates code for the underlined ones. Hereby the residual program shown to the right is produced. When a dynamic branch, *e.g.* an `if` is met, both the branches are specialized.

For specialization of statements a variant of *polyvariant program point* specialization is used [5]. The presence of descructive side-effects and pointers complicates the management of the specialization time stores considerably. "Dangling" pointers must be avoided aswell as destruction/introduction of sharing.

```
while (pp != STOP)
   switch(⟨Statement kind⟩)
      case ASSIGN: ⟨evaluate and update store⟩; pp += 1;
      case ASSIGN: gen_assign(⟨reduced expressions⟩); pp += 1;
      case GOTO: pp = ⟨goto label⟩;
      case GOTO: insert_pend(⟨goto label⟩); pp = STOP;
      case IF: if (⟨evaluate expression⟩)
                  pp = ⟨if then⟩ else pp = ⟨if else⟩;
      case IF: gen_if(⟨reduce expression⟩);
                     insert_pend(⟨if then⟩);
                     insert_pend(⟨if else⟩); pp = STOP;
      case RETURN:gen_return(⟨reduce expression⟩); pp = STOP;
      case CALL: ⟨perform static call⟩; pp += 1;
      case CALL: gen_call(⟨reduce arguments⟩);
                     insert_pend(pp+1); pp = STOP;
```

Fig. 1. Code generation for statements

A data structure `pending` keeps track of pending program points to be specialized. Of course, if a program point already have been specialized to a particular instance of the static values, it may be shared.

The drive loop is given in Fig. 2.

3.2 Function Specialization

Functions specialization is the generation of a residual function specialized with respect to static values, cf. `power_3()` above. In C, function specialization must obviously be with respect to both static parameters and static global variables. Function specialization is complicated due to the lack of referential transparency which characterize functional languages. It must be assured that all effects happens, and in the right order. Consider the following program.

```
int global; /*  A static global variable */
int main(void)
{ x = foo();
   S;
}
```

```
int foo(void)
{    if ( dynamic expression)
        global = 1;
      else
        global = 2;
     return  dynamic expression;
 }
```

```
spec_func(func, store)
{ insert_pend(⟨first label, store⟩); /* Insert first label */
   while (pending_not_empty())  /* Specializer all reachable points */
   { /* Restore computation state according to pending */
      ⟨ pp, store⟩ = pend_out();
      /* Specialize it */
      ⟨Fig.1⟩;
   }
   ⟨Store the generated code and record func has been specialized⟩;
}
```

Fig. 2. Drive loop for function specialization

Suppose `foo` is specialized. Clearly, function specialization must be *depth-first*, that is, `foo()` must be specialized before S is processed. Otherwise the (static) effects will happen in the wrong order. Imagine a call to `foo()` appears later in the program with the same values of the static variables. In a functional language, the call could immediately be replaced with a (residual) call to the already generated residual function, but in languages with side-effects it may be necessary to update the values of the static variables due to static side-effects in the specialized function. In C, the global and call-by-reference[1] must be updated according to the residual function.

Now consider the `global` variable in the program above. Even though it is assigned statically in `foo()`, its value will be unknown after the specialization. In the partial evaluator reported in [7], *explicators* are inserted and the binding time status of `global` changed to "unknown". Another solution which allows static treatment of `global`, and at the same time solves the problem of updating the store when residual functions are shared, is as follows.

```
int endconf;
int main(void)
{  x = foo();
    switch(endconf) {
      case 1: /* global=1 */
      case 2: /* global=2 */
 }
```

```
int foo()
{ if ( reduced expression) {
       endconf=1;
       return reduced expression;
     } else {
        endconf=2;
        return reduced expression;
    }
}
```

The key observation is that `global` only can assume finitely many static values[2], and hence S can be specialized with respect to these. The variable `endconf` is introduced at specialization-time, and its purpose is, at run-time, to indicate which code block to execute. Observe that the result of a function specialization is beside the generated code a set of specialization time store. These stores contain information for updating the callee after a residual call.

[1] Arrays are passed call-by-reference in C

[2] If the specialization process terminate

4 Experiments

An implementation of the C partial evaluator has been made. The specializer, `spec`, make up 472 lines of C code. The additional modules consist of app. 3000 lines of code. Benchmarks are given below. The time is user seconds on a Sun SparcStation II including parsing, and code size is number of C lines.

Program run	Code size		Time	
	Size	Ratio	Time	Ratio
$[\![scanner]\!](tokens, stream)$	72		1.6	
$[\![scanner_{tokens}]\!](stream)$	87	0.9	0.8	2.0
$[\![Int]\!](primes, 500)$	123		61.6	
$[\![Int_{primes}]\!](500)$	118	1.04	8.9	6.9
$[\![spec]\!](Int, primes)$	472		0.6	
$[\![spec_{int}]\!](primes)$	238	2.0	0.5	1.2

4.1 Specialization of a General Scanner

The general scanner from [9] accepts a token specification and a stream of characters, and output a list of recognized tokens. In the run, a specification of eight different tokens was given, and the scanner was applied to a file containing 21000 legal tokens.

By specialization of the scanner to the token specification, a lexical analysis is generated tailored to the particular set of tokens. The specialized program run approximately twice as fast as the general version. Gained by specialization is that all the "table lookups" are eliminated. The resulting program possesses a structure similar to a hand-written scanner.

The specialization time was 0.2 seconds. Hence, it even pay-off to specialize and run the specialized version when compared to the run-time of the general scanner.

4.2 Specialization of an Interpreter

We have implemented an interpreter for a "poolish-form" machine code language. The interpreter simulates a simple stack machine with instructions such as `JUMP`, `ADD`, and `STORE`. In the test run, input to the interpreter was a program that computes the first n primes.

The time for generation of the "polish-form"-compiler (by self-application of the specializer) was 1.8 seconds.

A speedup of 6.9 is obtained by specialization of the interpreter to the primes program. The result, a primes program in C, possess a machine code like form, and can be optimized considerably by the traditional code improving transformations in most C compilers.

The generated compiler is only halve the size of the general specializer. It does not, however, run faster than the specializer. This is mainly due to the overhead in the compiler—when larger programs are supplied, the overhead will disappear.

5 Related Work

This work continues the research in both program specialization of imperative languages and self-appplicable partial evaluation. To the best of our knowledge, this is the first developed and implemented self-applicable partial evaluator for a substantial imperative language.

Gomard and Jones reports a self-applicable partial evaluator for a small untyped flow-chart language with S-expression as the sole data structure [5]. In [6], a self-applicable partial evaluator for a subset of strongly typed LML is analyzed.

Meyer [7] describes an on-line partial evaluator for a subset of Pascal. The on-line technique allow variables to change binding time status during the specialization, but the off-line method may provide the specializer with valuable information which cannot be approximated "on-the-fly". A close study is beyond the scope of this paper. Nirkhe and Pugh [8] have developed an off-line system for a similar language.

6 Conclusion and Future Work

We have succeeded in developing, successfully implemented and self-applied a partial evaluator for a subset of C. The system has been applied in a number of experiments and give good results.

Several problems are still to be tackled, however. The foremost is treatment of general data structures such as pointers, lists and trees. Here the use of off-line static analyses may benefit as they can provide the specializer with valuable information.

References

1. L.O. Andersen. C program specialization. Technical report, DIKU, University of Copenhagen, Denmark, 1992.
2. L.O. Andersen. Self-applicable C program specialization. In *Proceeding of PEPM'92: Partial Evaluation and Semantics-Based Program Manipulation*, 1992.
3. A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.
4. A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, 1990.
5. C.K. Gomard and N.D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12:123–144, 1991.
6. J. Launchbury. A strongly-typed self-applicable partial evaluator. *Functional Programming Languages and Computer Architecture, August 1991. (LNCS, vol. 523)*, pages 145–164. ACM, Springer Verlag, 1991.
7. U. Meyer. Techniques for partial evaluation of imperative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 94–105. ACM, 1991.
8. V. Nirkhe and W. Pugh. Partial evaluation and high-level imperative programming languages with applications in hard real-time systems. In *Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992.* ACM, 1992.
9. F.G. Pagan. *Partial Computation and the Construction of Language Processors.* Prentice-Hall, 1990.