Self-Applicable C Program Specialization

Lars Ole Andersen
DIKU, Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagn Ø, Denmark
E-mail: lars@diku.dk

Abstract

A partial evaluator is an automatic program transformation tool. Given as input a general program and part of its input, it can produce a *specialized* version. If the partial evaluator is self-applicable, *program generators* can be made. The goal is *efficiency*: the specialized program often runs an order of magnitude faster than the general one.

We consider partial evaluation of the pragmatic oriented imperative C programming language. New problems studied includes partially static data structures, non-local static side-effects under dynamic control, and a restricted use of pointers. We define a Core C language, derive a two-level Core C language with explicit binding times, and formulate *well-annotatedness* conditions. Function specialization and code generation is described in terms of the two-level Core C language.

An implementation of the C partial evaluator has been made. Some experimental results are given.

1 Introduction

Partial evaluation is now a well-established tool for automatic program transformation and optimization. Given a *general* program and parts of its input data, a *specialized* version can be generated automatically. If the partial evaluator is *self-applicable*, program *generators* can be produced. The aim is *efficiency*: the specialized program often runs an order of magnitude faster than the original, general purpose program.

1.1 Background

A partial evaluator is a program operating on programs as data objects. This naturally imposes some restrictions on the subject languages that are treated, especially if the specializer is self-applicable. Previously, functional languages were used due to their simple semantics. The self-applicable partial evaluators reported in [7,3,6,4] all treat untyped functional languages with S-expressions as the sole data structure. In [5], a self-applicable partial evaluator for a small flow-chart language is described, but without functions and imperative data structures. Problems connected with partial evaluation of typed languages are studied in [9].

There are good reasons for self-applicable partial evaluation of an imperative, typed language. Many applications are most naturally expressed in imperative terms; imperative languages are usually more efficient on the computers; typed (abstract) data structures are nearly always more convenient than unstructured S-expressions, and last but not least: they are in widespread use. Further, it is a general feeling that types are good: they catch many errors at compile-time, they give more robust programs, and may be helpful to the programmer.

In this paper we study self-applicable partial evaluation of a substantial subset of the C programming language [8], including global variables and functions, data structures such as multi-dimensional arrays and structures, and almost all kind of statements and expressions.

Emerging problems connected with function specialization are non-local side-effects possibly under dynamic control, specialization with respect to pointers and aliases, and strategies. The treatment of the store at specialization-time is also considerably more difficult than in a functional world due to destructive side-effects and the presence of pointers. Self-application is complicated by the available data structures and the semantics of C.

This paper is essentially a summary of the work reported in [1], which is also available as [2]. In these reports, details of C program specialization can be found. We presuppose some knowledge in the field of self-applicable partial evaluation, *e.g.* corresponding to the papers [7,5], and of the semantics of the C programming language [8].

1.2 Overview

In Section 2 we recall the foundation of partial evaluation with focus on *types*. Section 3 introduces Core C, which is an intermediate language. Core C captures the essential parts of C, and is the language the partial evaluator treats. Subject for Section 4 is separation of the binding times. We derive a two-level Core C language where binding times are explicit in the syntax. Section 5 contain descriptions of the basic techniques applied, hereunder treatment of non-local side-effects, pointers, aliases, and function specialization. An implementation of parts of the C partial evaluator has been made. In Section 6 we provide some experimental results and assessments. Section 7 discusses related work, and Section 8 gives several directions for future work, and concludes.

2 Partial Evaluation and Types

A partial evaluator treat programs as data objects. It is therefore important to distinguish between a *program text*, a *representation* of a program, and the *meaning* of a program. Let p be the text of a program; we will usually take p to be in the C language. The meaning of p can be regarded as a function from the input parameters to the return value (when no input is through external variables *etc*). The execution of p (on a C machine) on input $d_1, \ldots, d_n$ is written $[\![p]\!]_C(d_1, \ldots, d_n) \Rightarrow d$, where d is the result (if any).

Suppose `sint` is a self-interpreter for C, taking as input a program p and its input data d_i, and yield as result a value d as normal execution of p would. In languages like Scheme and Lisp, programs are legal data objects but in C, a suitable *representation* must be devised. By $\overline{p}^{\mathtt{Pgm}}$ we denote an (unspecified) representation of p in a data structure of type `Pgm`. Consider the d_i. Since the self-interpreter is intended to work on all programs— which may take input data of different types—the data d_i must be encoded into a single union type `Val`. In C, `Val` can be a `struct` with a huge `union` inside. Base types (`int`, `char`, ...) can be represented by themselves; an array can be represented by a pointer to an externally allocated array[1], and similarly for structures.[2] Hence, applying the self-interpreter to p is properly denoted by:

$$[\![\mathtt{sint}]\!]_C(\overline{p}^{\mathtt{Pgm}}, \overline{d_1}^{\mathtt{Val}}, \ldots, \overline{d_n}^{\mathtt{Val}}) \Rightarrow \overline{d}^{\mathtt{Val}}$$

where the result also is delivered in the `Val` type.

EXAMPLE 2.1 Let `pow` be the well-known power program computing x to the nth. Applying the self-interpreter to `pow` is thus equivalent to: $[\![\mathtt{sint}]\!]_C(\overline{\mathtt{pow}}^{\mathtt{Pgm}}, \overline{2}^{\mathtt{Val}}, \overline{3}^{\mathtt{Val}}) \Rightarrow \overline{8}^{\mathtt{Val}}$, where the result 8 (an integer) must be projected out of the union type. END OF EXAMPLE

Consider self-application of the self-interpreter, giving the interpreted version of `sint` the input p and d. As an input to `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. In other symbols:

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

where program and input data has been double encoded. The problem is: during self-application a huge data structures will exist and impede efficient self-application. Launchbury made the same observation during the development of a partial evaluator for a subset of LML. [9]. A way to reduce the size of the data structures is to included `Pgm` in `Val` as a *new base type*. For a closer study of program and value representation we refer to [2].

The *Mix Equation* can now be formulated as follows.

THEOREM 2.1 (THE MIX EQUATION)
Let p be a program taking input d_i. A partial evaluator `mix` *is a program such that if $[\![p]\!]_C(d_s, d_d) \Rightarrow d$, then*

$$[\![\mathtt{mix}]\!]_C(\overline{p}^{\mathtt{Pgm}}, \overline{d_s}^{\mathtt{Val}}) \Rightarrow \overline{p_{d_s}}^{\mathtt{Pgm}} \text{ and } [\![p_{d_s}]\!]_C(d_d) \Rightarrow d$$

provided `mix` *terminates.* □

[1] Since the size of the array can be arbitrary, the array can not reside inside Val

[2] Care must be taken. Structures are passed call-by-value in C

As usual, the part of the input given to `mix` is called the *static* input; the rest the *dynamic* input. The `mix` produced program is called the residual program and is a specialized version of p with respect to d_s.

The possibility of compilation, generation of stand-alone compilers, and even a compiler generator is expressed in the Futumura projections. These are restated below with explicit representation of programs and values.

THEOREM 2.2 (THE FUTAMURA PROJECTIONS) *Let* `int` *be an interpreter interpreting p programs. Suppose* `mix` *is a partial evaluator. Then the following holds:*

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

on the assumption that the executions terminate. □

Another look at partial evaluation is as a *non-standard* execution of the program. Let p^{ann} be a binding time annotated program. Then partial evaluation corresponds to an *execution* of p^{ann} using a *partial evaluation semantics*:

$$[\![p^{ann}]\!]_{C_{pe}}(\overline{d_s}^{\mathtt{Val}}) \Rightarrow \overline{p_{d_s}}^{\mathtt{Pgm}}$$

where C_{pe} is a "C partial evaluation language". This is the view we take in this paper.

3 The Core C Language

A programming language subject for (self-applicable) partial evaluation should ideally meet two contrary goals. From a pragmatic viewpoint, a rich language with many constructs and data structures is desirable. On the other hand, to achieve self-application, a modest sized language is definitely required.

Core C is a control-flow representative of the C programming language. Control-flow is explicit via jumps and function calls. There are no control structures as *e.g.* `while`. Since program specialization essentially is the specialization of *control-flow* to static values, this is profitable. The abstract syntax of Core C is given in Figure 1.

A Core C program consists of an optional number of global variable declarations followed by at least one function definition. It is not possible to restrict the scope of global variables as in C. A function can declare both parameters and local variables. Block scope declaration is not possible, but this can obviously be solved automatically during transformation from C to Core C.

The supported data structures include base type variables, structures, multi-dimensional arrays, and pointers to arrays. General pointers to *e.g.* heap allocated objects and functions, are not handled.

The possible statements are asignment, function return, unconditional jump, conditional jump, and function call. The meaning of the `call` statement is as follows. The actual parameters are evaluated and the function called. The returned value is assigned the variable appearing on the left-hand side of a call. Observe that both assignments and function calls are on the statement level in Core C. Hereby evaluation of expressions cannot cause side-effects to occur, which is convenient for implementation purposes, cf. Section 5.

id, fid,eid	:	Identifiers, function identifiers and external function identifiers	
basetype	:	Base types (`int`, `double`, *etc*)	
label	:	Labels	
const	:	Constants (integer, double, *etc*)	
uop, bop	:	Unary and binary operators	
CoreC	::=	vardecl* fundef$^+$	*Core C program*
vardecl	::=	type *id* typespec	*Variable declarations*
type	::=	*basetype* | type '$*$' | `struct` *id* '{' vardecl$^+$ '}'	
typespec	::=	ϵ | typespec '[' *const* ']'	
fundef	::=	type *fid* '(' vardecl* ')' '{' vardecl* stmt$^+$ '}'	*Function definitions*
stmt	::=	*label*: `assign` lval '=' exp	*Statements*
	|	*label*: `return` exp	
	|	*label*: `goto` *label*	
	|	*label*: `if` '(' exp ')' *label label*	
	|	*label*: `call` *id* '=' *fid* '(' exp* ')'	
exp	::=	`const` *const* | `var` *id*	*Expressions*
	|	`index` exp '[' exp ']' | `struct` exp '.' *id*	
	|	`uop` *uop* exp | `bop` exp *bop* exp | `ecall` *eid* '(' exp* ')'	
lval	::=	*id* | lval '[' exp ']'	*Left-expressions*

Figure 1: Abstract syntax of Core C

An expression can be a constant, a variable reference, an array or structure indexing, an application of a unary or binary operator, or an application of an external function. The address operator & is supported, but as only pointers to arrays are allowed, it may only be applied to expressions evaluating to such.

Obviously, almost full Ansi C can be transformed into a corresponding Core C program, the major absence being heap allocated and pointer linked data structures. This way the partial evaluator can handle almost the full language, but the specializer needs only to deal with the Core C subset. In [2] the dynamic semantics for Core C is specified via an operational semantics.

4 Separation of Binding Times

Program specialization can roughly be considered as a non-standard execution. A statement depending solely on available static information is executed normally; code is generated for the rest. Hence, for each particular statement, the specializer must hence know whether to execute or generate code. The separation of binding times can essentially be done in two stages: either *before* the specialization ("off-line"), or *during* the specialization ("on-line"). On-line techniques have been advocated for imperative programs [10], but for achieving self-application, experience clearly demonstrates that off-line methods are superior. We will exclusively make use of off-line specialization, but not in this paper consider binding time analysis. An efficient binding time analysis based on type inference is described in [2].

4.1 Two-Level Core C

In the two-level Core C language, binding times are explicit in the syntax. Every construct which can be either static or dynamic appears in two versions: a (normal) static one, and a (underlined) dynamic. The syntax of the two-level Core C language is derived from the abstract syntax of Core C (Figure 1) by addition of underlined versions of types (for variable separation), expressions, and statements. For example, the assignment:

```
1:  assign lval = exp
```

denotes a specialization-time assignment, and

```
1:  assign lval = exp
```

represents a dynamic assignment. Similarly for expressions, *e.g.* `bop` is a static expression, while `bop` is dynamic.

EXAMPLE 4.1 Consider again the `power` program. Suppose n is static and x is dynamic. Below the two-level Core C version can be seen.

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

Since n is static, the computation and control-flow involving n can be done at specialization time. On the other hand, `pow` and the assignments to it must be dynamic, since the value of x is unknown. The return statement must be dynamic for two reasons: `pow` is a dynamic variable, and `power()` is a residual function which must return "something" at runtime. END OF EXAMPLE

Consider the assignment `pow = 1` in the example above. The variable `pow` is dynamic, but the constant "1" is clearly static. This is a "static computation in a dynamic context". For making such shift in the binding time explicit, a *lift* operator is introduced: `1: assign pow = lift 1`. The meaning is: "evaluate the static expression and built a residual constant". We impose the following restriction upon the use of `lift`: it may only be applied to base-type expressions. For instance, arrays may not be "lifted".

4.2 Well-Annotatedness of two-level Core C

Not every two-level program can be specialized. For instance, the indispensable requirement "a static expression may not depend on a dynamic expression" is not captured by the syntax of two-level Core C. A two-level Core C program satisfying these additional conditions is said to be *well-annotated*.

The binding time of an expression can be seen as a *type* in the two-level language.

Definition 4.1 (Binding Time Types)
The syntax of a two-level binding time type is given by:

$$\mathcal{T} ::= S \mid D \mid [\mathcal{T}] \mid \mathcal{T} \otimes \mathcal{T}$$

where S and D are ground types. □

The types S and D denote static constant and completely dynamic value, respectively. The constructor $[\mathcal{T}]$ denote an array *statically* indexed with elements of type $\mathcal{T}$. An array which is not overall statically indexed is assigned the type D. The product constructor $\otimes$ describe structures. For notational convenience we will also freely use the product $\times$ for describing the type of functions *etc.* Observe that the `lift` operator possesses the binding time type $\texttt{lift} : S \to D$ and represents thus a (dynamic) coercion function.

A binding time type assignment $\tau : Id \to \mathcal{T}$ is a finite map from identifiers to binding time types. We assume that the map $\gamma : Id \to \mathcal{T}$ assign (static) binding time types to operators and external functions, *e.g.* $\gamma(+) = S \times S \to S$. An assignment of types to the main parameters is called an *initial* division, and an assignment which agree on the main parameters with an initial division is said to *respect* the initial division.

Well-annotatedness of expressions can thus be seen as a matter of *well-typing*. A typing of form $\tau, \gamma \vdash e : \mathcal{T}$ reads: "given type assignment τ and γ, the expression e (statement) possess the type $\mathcal{T}$. Suppose an initial division τ_0 is given.

Definition 4.2 (Well-Annotatedness Expressions)
Given $\tau, \gamma : Id \to \mathcal{T}$ respecting τ_0. An expression e is well-annotated if there exist a two-level type $\mathcal{T}$ such that $\tau, \gamma \vdash^{exp} e : \mathcal{T}$, where relation $\vdash^{exp}$ is defined in Figure 2. □

A constant is static and an application of the lift operator to a static expression yields a dynamic expression. The type of variables is given by the type assignments τ.

If the left expression e_1 in an array index expression $e_1[e_2]$ possess the type $[\mathcal{T}]$ and the index expression e_2 is static, the whole expression has type $\mathcal{T}$. Otherwise all parts must be dynamic and hence the indexing deferred to runtime. We shall see that this is tightly connected with partially static data structures and specialization-time splitting of these. Similar holds for structure index expressions.

The rules for unary, binary and external operators are as expected. Observe that if one of the arguments is non-static, all arguments must be dynamic and the application suspended. The rule for the address operator forces all such applications to be dynamic. This ease the implementation of the memory management in the specializer, cf. [2].

Well-annotatedness of statements can be expressed in a similar way using the two-level types C and R. The type C denote "compile-time" and R "run-time".

Definition 4.3 (Well-Annotatedness of Statement)
Let $\tau, \gamma : Id \to \mathcal{T}$ be type assignments respecting the initial division, and let $\pi : Id \to \mathcal{T}$ be a function environment (describing the return value of user defined functions). Statement s is well-annotated if there exist a type T such that $\tau, \gamma, \pi \vdash^{stmt} s : T$, where $T \in \{C, R\}$. □

The formal definition of the relation $\vdash^{stmt}$ is omitted due to lack of space; it can be found in [2]. The typings for assignments are as follows.

$$\frac{\tau, \gamma \vdash^{exp} lval : T \quad \tau, \gamma \vdash^{exp} exp : T \qquad \text{T static}}{\tau, \gamma, \pi \vdash^{stmt} \texttt{assign}\ lval = exp : C}$$

where "T static" means "D does not occur in T", and

$$\frac{\tau, \gamma \vdash^{exp} lval : D \quad \tau, \gamma \vdash^{exp} exp : D}{\tau, \gamma, \pi \vdash^{stmt} \underline{\texttt{assign}}\ lval = exp : R}$$

for run-time assignments.

The rules for function calls are slightly more complicated due to the presence of non-local static side-effects.

4.3 Partially Static Data Structures

A data structure containing dynamic values, but where parts of it are static, is called a *partially static data structure*. In certain conditions, a partially static data structure may be split during the specialization into separate variables. For example, the array

```
int a[3]
```

can in some cases be split into the variables

```
int a_0, a_1, a_2;
```

and an expression `a[`e`]` replaced by `a_1` (assuming e evaluates to 1). Advantages hereof include register allocation, direct access *etc.*

In the functional world, splitting of partially static structures is known as *arity raising*. The semantics of C complicate matters considerably.

Example 4.2 Assume the array `a` in the program below is split and passed as separate variables to the function `foo()`.

```
int a[2];
int foo (int x[])
{
   x[1] = 2;
   return x[0];
}
```

Recall that arrays are passed call-by-reference in C. By splitting it, the side-effect will not be observable outside `foo()`, *i.e.* the semantics has changed. End of Example

Splitting an array is semantically *safe* if the array is *splittable* according to the following definition.

Definition 4.4 (Splittable Array)
An array is non-splittable *if:*

- *it is passed to a function, or*
- *it is returned by a function, or*
- *it is passed to an external function*

$$\tau, \gamma \vdash^{exp} \texttt{const}\ c : S \qquad \frac{\tau, \gamma \vdash^{exp} e : S}{\tau, \gamma \vdash^{exp} \texttt{lift}\ e : D}$$

$$\frac{\tau(v) = T}{\tau, \gamma \vdash^{exp} \texttt{var}\ v : T} \qquad \frac{\tau(x) = D}{\tau, \gamma \vdash^{exp} \underline{\texttt{var}}\ x : D}$$

$$\frac{\tau, \gamma \vdash^{exp} e_1 : [T] \quad \tau, \gamma \vdash^{exp} e_2 : S}{\tau, \gamma \vdash^{exp} \texttt{index}\ e_1[e_2] : T} \qquad \frac{\tau, \gamma \vdash^{exp} e_1 : D \quad \tau, \gamma \vdash^{exp} e_2 : D}{\tau, \gamma \vdash^{exp} \underline{\texttt{index}}\ e_1[e_2] : D}$$

$$\frac{\tau, \gamma \vdash^{exp} e : T_x \otimes \cdots \otimes T_y \otimes \cdots \otimes T_z}{\tau, \gamma \vdash^{exp} \texttt{struct}\ e.y : T_y} \qquad \frac{\tau, \gamma \vdash^{exp} e : D}{\tau, \gamma \vdash^{exp} \underline{\texttt{struct}}\ e.y : D}$$

$$\frac{\tau, \gamma \vdash^{exp} e : D}{\tau, \gamma \vdash^{exp} \underline{\texttt{uop}}\ \&\ e : D}$$

$$\frac{\tau, \gamma \vdash^{exp} e_1 : T_1 \quad \gamma(op) = T_1 \to T}{\tau, \gamma \vdash^{exp} \texttt{uop}\ op\ e : T} \qquad \frac{\tau, \gamma \vdash^{exp} e : D}{\tau, \gamma \vdash^{exp} \underline{\texttt{uop}}\ op\ e : D}$$

$$\frac{\tau, \gamma \vdash^{exp} e_1 : T_1 \quad \tau, \gamma \vdash^{exp} e_2 : T_2 \quad \gamma(op) = T_1 \times T_2 \to T}{\tau, \gamma \vdash^{exp} \texttt{bop}\ op\ e_1\ e_2 : T} \qquad \frac{\tau, \gamma \vdash^{exp} e_1 : D \quad \tau, \gamma \vdash^{exp} e_2 : D}{\tau, \gamma \vdash^{exp} \underline{\texttt{bop}}\ op\ e_1\ e_2 : D}$$

$$\frac{\tau, \gamma \vdash^{exp} e_1 : T_1 \quad \cdots \quad \tau, \gamma \vdash^{exp} e_n : T_n \quad \gamma(f) = T_1 \times \cdots \times T_n \to T}{\tau, \gamma \vdash^{exp} \texttt{ecall}\ f(e_1, \ldots, e_n) : T} \qquad \frac{\tau, \gamma \vdash^{exp} e_1 : D \quad \cdots \quad \tau, \gamma \vdash^{exp} e_n : D}{\tau, \gamma \vdash^{exp} \underline{\texttt{ecall}}\ f(e_1, \ldots, e_n) : D}$$

Figure 2: Well-annotatedness rules for expressions

An array is splittable *if it is overall statically indexed (i.e. possess a two-level type $[T]$), and is not non-splittable.* □

It is crucial that the array is overall statically indexed—otherwise the index expression cannot be replaced by the proper variable. The three requirements can be justified as follows. If an array if passed as a parameter to a function, a change in the semantics is risked. If an array (pointer) is returned by a function it is obviously not possible to split the array. Finally, an external function is a "black-box" from the specializer's viewpoint: its type is fixed and may not be changed. A separate post-phase analysis may be able to split variables not captured by *splittable*, for example, an array which is dead after a function call may perfectly well be split and passed as separate variables, since possible side-effects in the called function will not be observable anyway.

Example 4.3 Notice that the typing rule for dynamic assignments forces the type of the assigned expression to be D. If *e.g.* the type $[D]$ was allowed, an assignment `assign x = a` (where `a` is an array), would have to be replaced by the sequence of assignments

```
x_0 = a_0; x_1 = a_1; ...
```

This is inconvenient for several reasons, and is hence prohibited. End of Example

Splitting of structures can be defined in a similar way.

A particular important aspect of variable splitting is *optimality* of the specializer. If the array representing a store in an interpreter is not split, the residual program will represent all values indirectly, and optimality is lost. This is analyzed in detail in [2], and it is found that several other "tricks" are needed for obtaining optimality.

5 Specialization of Core C

For specialization for Core C we apply an adapted version of *polyvariant program point specialization* as *e.g.* described in [5,3]. There are two kind of program points: functions and target of jumps. By function specialization we mean the specialization of a function to (part) of the static values. New problems encountered here are pointers, aliases, and non-local side-effects.

5.1 Function Specialization and non-local Side-Effects

Obviously, function specialization must be with respect to both static *parameters* and static *global variables*. Better sharing will occur if specialization is only to *live* variables. Care must be taken not to destroy side-effects by sharing of residual functions.

In C, referential transparency is not valid. This influences the order in which function specialization must be done.

Example 5.1 Consider the program below. Suppose that `foo()` is to be specialized due to the call in `main()`.

```
int global;
int main()                    int foo (int x)
{                             {
    1: call x = foo(y)            1: assign global = 1
    2: assign z = global          2: return x + x
}                             }
```

Be cause of the static side-effect on `global` (which is used in `main()` after the function call) `foo()` must be dealt with *before* the succeeding statements. End of Example

Observation 5.1 In an imperative language with non-local static side-effects, function specialization must be performed *depth-first*. End of Observation

In a functional language, the order of function specialization is immaterial since the processing cannot affect the specialization of the remaining expressions.

The non-local, but static side-effects in a function may be under *dynamic* control, as illustrated by the following program.

Example 5.2 The variable `global` is assigned statically in `foo()`.

```
int global;
int main()                int foo (int x)
{                         {
  1: call x = foo(y)        1: if (x) 2 4
  2: /* Global? */          2: assign global = -1
                            3: return  exp1
```

```
                    4: assign global = 1
                    5: return  exp2
}                 }
```

End of Example

Even though the assignments to `global` are static, the concrete value of it is unknown after the call. To alleviate the problem, such dynamically controlled non-local side-effects could be banned, or explicators inserted [10]. Two other possibilities are: *unfolding* and *finitely many static value* specialization. In order to deal with (mutually) recursive functions, some dynamically controlled static variables must be made dynamic, though.

Example 5.3 On-line unfolding of `foo()` into `main()` will yield the following program, which can be specialized without problems.

```
int global;
int main()
{
    1: if (y) 2 5
    2: assign global = -1
    3: assign x =  exp1
    4: goto 8
    5: assign global = 1
    6: assign  x =  exp2
    7: goto 8
    8: ...
}
```

Notice that the polyvariant specialization strategy to be presented below, allows sharing of program point 8 if the static values are the same. End of Example

Infinite unfolding of recursive functions must be prevented forcing (some) non-local side-effects to be suspended.

Another strategy is based on the following observation. If specialization terminates, the static variables can only assume *finitely* many values. The basic idea is to specialize the statements following a call with respect to all the possible values of the side-effected variables.

Example 5.4 A residual program for the above example. The variable `endconf` is introduced at specialization-time.

```
int endconf;
int main()                    int foo()
{                             {
  1: call x = foo(y)            1: if (y) 2 4
  2: if (endconf == 1) 4 3      2: assign endconf = 1
  3: if (endconf == 2) 5 0      3: return  exp'1
  4: /* global == -1 */         4: assign endconf = 2
  5: /* global == 1  */         5: return  exp'2
}                             }
```

The variable `endconf` is a "continuation". End of Example

Here, function specialization gives rise to both generation of code and a set of *specialization-time stores* recording the values of the static variables. Implementation details are given in [2]. Note that the accumulated stores contains the side-effects of the function.

Both strategies can handle the typical use of dynamically controlled non-local static side-effects. For example, a function which in some condition "sets a flag", or a scanner reporting "end-of-file". In the implementation we have made use of the finitely-many-values strategy.

5.2 Pointers and Aliases

In Core C, pointers are to arrays, and can originate from the passing of an array as a parameter or an application of the address operator `&`. In a functional language, two lists are considered equal if they equal element-wise even though they may be allocated in different locations in the heap.

Example 5.5 Assume that the array $\mathtt{a} = \{1, 2, 3\}$ and $\mathtt{b} = \{1, 3, 3\}$ are both static.

```
int a[10], b[10];
int main()                    int foo(int *x)
{                             {
   1: call x = foo(a)            1: assign x[2] = 2
   2: call x = foo(b)            2: return lift b[2]
}                             }
```

One could imagine that the residual function `foo_a()` could be shared at the residual call `foo(b)` (notice that the principle of specialization to static variables is not violated). This is wrong, as easily can be seen. End of Example

Observation 5.2 Specialization must be with respect to both the *address* and the *indirection*. End of Observation

The presence of pointers complicates the management of the store considerably. At a specialization point, the store must be copied since it is updated destructively. Doing this, "dangling" pointers must be avoided. In an implementation of the C partial evaluator, we have exploited a copy/restore memory management scheme. The restriction that applications of & are dynamic, is mainly for allowing easy copying and comparison of specialization-time store. It is part of the binding time analysis to assure that *e.g.* a dynamic pointer does not point into a static structure (which will not be present in the residual program).

5.3 Two-Level Semantics for Statements

In this section we briefly present the processing of statements. The core of the *polyvariant* driver loop is shown below.

```
pend = {⟨1, σ⟩}; done = { };
while (pend ≠ { })
   ⟨l, σ⟩ = pick element from pend and remove it;
   done = done ∪ {⟨l, σ⟩};
   specialize program point l to store σ
```

The specialization of statements is given in form of a transition relation:

$$\vdash^{stmt}: \text{Stmt} \times \text{Store} \rightarrow \text{Label} \times \text{Stmt} \times \text{Store}.$$

It is a function from a statement and a (specialization-time) store to a label ("next label"), a (residual) statement, and a (possibly updated) store.

The relation can be seen as a definition of the *semantics* of two-level Core C statements. Whereas normal execution yields a "next-label" and a store, two-level execution yields a set of "next-labels"[3], a specialization-time store, and some code.

Let $\rho : Id \rightarrow$ Location be an environment function mapping identifiers to locations. The store σ : Location $\rightarrow$ Values is a function from locations to values. We use the

[3] processing of a dynamic if gives rise to two labels

$$\frac{\rho,\sigma \vdash^{exp} lval \Rightarrow loc \quad \rho,\sigma \vdash^{exp} exp \Rightarrow v}{\rho \vdash^{stmt} \langle l : \texttt{assign}\ lval\ \texttt{=}\ exp, \sigma\rangle \Rightarrow \langle l', \cdot, \sigma[v/loc]\rangle}$$

$$\frac{\rho,\sigma \vdash^{exp} lval \Rightarrow lval' \quad \rho,\sigma \vdash^{exp} exp \Rightarrow exp'}{\rho \vdash^{stmt} \langle l : \texttt{assign}\ lval\ \texttt{=}\ exp, \sigma\rangle \Rightarrow \langle l', \langle l, \sigma\rangle : \texttt{assign}\ lval'\ \texttt{=}\ exp', \sigma\rangle}$$

$$\rho \vdash^{stmt} \langle l : \texttt{goto}\ m, \sigma\rangle \Rightarrow \langle m, \cdot, \sigma\rangle$$

$$\rho \vdash^{stmt} \langle l : \underline{\texttt{goto}}\ m, \sigma\rangle \Rightarrow \langle \cdot, \langle l, \sigma\rangle : \text{goto}\ \langle m, \sigma\rangle, \sigma\rangle$$

$$\frac{\rho,\sigma \vdash^{exp} exp \Rightarrow v \neq 0}{\rho \vdash^{stmt} \langle l : \texttt{if}\ (exp)\ m\ n, \sigma\rangle \Rightarrow \langle m, \cdot, \sigma\rangle}$$

$$\frac{\rho,\sigma \vdash^{exp} exp \Rightarrow 0}{\rho \vdash^{stmt} \langle l : \texttt{if}(exp)\ m\ n, \sigma\rangle \Rightarrow \langle n, \cdot, \sigma\rangle}$$

$$\frac{\rho,\sigma \vdash^{exp} exp \Rightarrow exp'}{\rho \vdash^{stmt} \langle l : \underline{\texttt{if}}(exp)\ m\ n, \sigma\rangle \Rightarrow \langle \cdot, \langle l, \sigma\rangle : \texttt{if}\ (exp')\langle m, \sigma\rangle\ \langle n, \sigma\rangle, \sigma\rangle}$$

$$\frac{\begin{array}{l}\rho,\sigma \vdash^{exp} exp_1 \Rightarrow v_1 \cdots \rho,\sigma \vdash^{exp} exp_n \Rightarrow v_n \\ \rho \circ [l_1/x_1, \ldots, l_n/x_n] \vdash^{func} \langle f, \sigma[v_1/l_1, \ldots, v_n/l_n]\rangle \Rightarrow \langle v, \sigma'\rangle\end{array}}{\begin{array}{l}\rho \vdash^{stmt} \langle l : \texttt{call}\ x = f(exp_1, \ldots, exp_n), \sigma\rangle \Rightarrow \\ \qquad \langle l', \cdot, \sigma[v/\rho(x)]\rangle\end{array}}$$

$$\frac{\begin{array}{l}\rho,\sigma \vdash^{exp} exp_1 \Rightarrow v_1 \cdots \rho,\sigma \vdash^{exp} exp_i \Rightarrow v_i \\ \rho,\sigma \vdash^{exp} exp_{i+1} \Rightarrow exp'_{i+1} \cdots \rho,\sigma \vdash^{exp} exp_n \Rightarrow exp'_n \\ \rho \circ [l_1/x_1, \ldots, l_i/x_i] \vdash^{func} \langle f, \sigma[v_1/l_1, \ldots, v_i/l_i]\rangle \Rightarrow \langle f', \{\langle 1, \sigma_1\rangle, \ldots, \langle m, \sigma_m\rangle\}\rangle\end{array}}{\begin{array}{l}\rho \vdash^{stmt} \langle l : \underline{\texttt{call}}\ x = f(exp_1, \ldots, exp_n), \sigma\rangle \Rightarrow \\ \qquad \langle \cdot, \langle l, \sigma\rangle : \texttt{call}\ x = f'(exp'_{i+1}, \ldots, exp'_n), \cdot\rangle\end{array}}$$

$$\frac{\rho,\sigma \vdash^{exp} exp \Rightarrow exp'}{\rho \vdash^{stmt} \langle l : \underline{\texttt{return}}\ exp, \sigma\rangle \Rightarrow \langle \cdot, \langle l, \sigma\rangle : \texttt{return}\ exp', \sigma\rangle}$$

Figure 3: Two-level semantics for statements

notation $\langle l, \sigma\rangle$ to denote a run-time label generated during specialization (program point l specialized to store σ).

The semantics for two-level statements is given in Figure 3. Evaluation of static expressions is formalized via the relation $\vdash^{exp}$: Exp $\rightarrow$ Value. The label l' is consistently used to mean the label of the statement following the l labelled statement.

Static statements are executed as normal.

In the case of a dynamic assignment, both the expressions are reduced and a residual assignment built. After the processing of a dynamic goto or if, the target labels must be inserted into `pend` for specialization. Of course, if a program point already has been specialized to the particular instance of the static variables, the residual code can be shared.

Consider now a dynamic call to a function, where the function is to be specialized. The static arguments are evaluated, and the dynamic ones reduced. If a corresponding call has been met before, the (already generated) residual can be shared. Otherwise the function must be specialized. The function specialization yields a set of specialization-time stores recording the side-effects that happened in the function. The program point following the call must be specialized with respect to the store (in the callee) updated with the effects recorded in the collected stores. Finally, code for the branch on the `endconf` variable must be generated.

In the case of a dynamic `return` statement, an assignment to the `endconf` variable must be generated, aswell as a return statement. A (copy) of the current store must be added to the collection of *end-configuration stores*, to be returned to the callee.

In [2] a complete specification of the two-level semantics can be found.

In an implementation it can be exploited that evaluation of expressions cannot cause side-effects which means it can be done externally, *e.g.* via an external function call `eval_exp()`. As the specializer does not have to "look" at the internals of an expression, it can be represented using data structures not treated by the system, *e.g.* pointer linked structures. In our implementation, a new basetype `Expr` has been introduced. The specializer knows nothing about the internals of an `Expr` expression.

6 Experiments

We have made an implementation of the C program specializer and applied it in a number of experiments. Currently, a binding time analysis has not been implemented, and thus the system is not yet fully automatic. In [2] a binding time analysis based on constraint set solution is described. Besides the binding time analysis, an *untagging* analysis is needed for simplifying the types in the residual programs. Consider for example specialization of a self-interpreter to a program. Since the residual program inherits its types from the self-interpreter, it will use `Val` variables for representation of *e.g.* integer variable. The purpose with the untagging analysis is to detect this and simplify the types when possible.

Below we present the result of two experiments. Both are from [13], and hence demonstrate that automation is possible.

A General Scanner

A general scanner is a program taking as input a *token specification* and a stream of characters. By specializing the scanner to the token table, a specific scanner will be generated tailored to the given set of tokens.

Program	Time	Ratio	Code size
$[\![Scanner]\!](Token, Stream)$	1.5		74
$[\![Scanner_{res}]\!](Stream)$	0.9	1.7	122

The scanner was applied to a specification of 8 tokens, and recognized 30000 tokens. The time is CPU seconds, and the code size is lines of C code.

A Postfix Interpreter

We have specialized an interpreter for a postfix language. The program given to the interpreter output the first n primes. Below the results can be seen.

Program	Time	Ratio	Code size
$[\![Int]\!](primes, 500)$	60.0		122
$[\![Int_{primes}]\!](500)$	8.9	6.7	139
$[\![spec]\!](int, primes)$	5.8		611
$[\![spec_{int}]\!](primes)$	0.5	11.6	618

The input 500 to the *primes* programs causes the first 500 primes to be generated. The residual program (in C computing primes) has not been optimized in any way. There are several opportunities for optimization. The compiler generation time was 11.5 seconds, and cogen is produced in 29.7 seconds. The figures does not include postprocessing.

7 Related Work

The present work continues the work in self-applicable partial evaluation and specialization of imperative languages. To the best of our knowledge, this is the first self-applicable partial evaluator for an imperative language which includes both global variables and functions, imperative data structures, and the usual statements and expressions. We would also like to stress that the partial evaluator treats a rather large language—it is not a "toy" language!

Pagan describe several "hand-methods" for conversion of general program into generating programs [13]. Our work automates the techniques of Pagan.

Meyer [10] reports a partial evaluator for subset of Pascal. Integer variables are the only treated kind of data structure. The use of on-line specialization allows his system in some cases to do more at specialization time than ours. We believe that *livenesse* and introduction of new variables may alleviate these differences. Static variables under dynamic control are handled by the use of *explicators*. This effectively forces such variables to be dynamic. Our technique only makes dynamically controlled variables dynamic, when the control is in a recursive function—the price being larger residual programs.

Nirkhe and Pugh [12] describe a partial evaluator for specialization of hard real-time problems. In order to guarantee run-time properties, they impose some constraints upon the programs, but the basic techniques are similar to Meyers. Neither Meyer nor Nirkhe & Pugh's partial evaluators are self-applicable, and they cannot treat data structures such as arrays and structures, nor split partially static data structure.

Gomard and Jones report a self-applicable partial evaluator for a small flow-chart language over the set of S-expressions. There are no functions and the use of untyped S-expressions eliminate problems with representation of programs and values. The polyvariant specialization technique we have applied is similar to the one used there.

Lambda-mix is a self-applicable partial evaluator for the untyped lambda calculus [6]. The use of two-level languages and type inference for describing well-annotatedness were founded [11], and applied in this project. However, the data structure and semantics of our language is considerably more complicated than that of lambda calculus.

8 Conclusion and Future Work

We have developed a self-applicable partial evaluator for a substantial subset of the C programming language. Problems with representation of values, non-locally destructive side-effects, pointers, and partially static data structures are addressed. An implementation of part of the system have been made. Some experimental results were given.

There are several directions for future work. The treated kind of data structures should be extended to include pointer linked structures. This will require some kind of alias analysis in connection with binding time analysis. Next, methods for better static processing should be investigated. For example, polyvariant binding time analysis might allow a function to be specialized to several different static/dynamic call configurations. Finally, practical experiments of partial evaluation is needed. We believe that the use of the C language will open up for many new applications of partial evaluation.

Acknowledgements

Thanks to Torben Mogensen for useful comments.

References

[1] L.O. Andersen. C program specialization. Master's thesis, DIKU, University of Copenhagen, Denmark, December 1991. DIKU Student Project 91-12-17, 134 pages.

[2] L.O. Andersen. C program specialization. Technical Report 92/14, DIKU, University of Copenhagen, Denmark, May 1992. Revised version.

[3] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. Technical Report 90/4, DIKU, University of Copenhagen, Denmark, 1990.

[4] C. Consel. New insights into partial evaluation: The Schism experiment. In H. Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 1988. (Lecture Notes in Computer Science, vol. 300)*, pages 236–246. Springer Verlag, 1988.

[5] C.K. Gomard and N.D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12:123–144, 1991.

[6] C.K. Gomard and N.D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[7] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[8] B.W. Kernighan and D.M. Ritchie. *The C programming language (Draft-Proposed ANSI C)*. Software Series. Prentice-Hall, second edition edition, 1988.

[9] J. Launchbury. A strongly-typed self-applicable partial evaluator. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 145–164. ACM, Springer Verlag, 1991.

[10] 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.

[11] F. Nielson and H.R. Nielson. the tml-approach to compiler-compilers. Technical Report 1988-47, Department of Computer Science, Technical University of Denmark, 1988.

[12] 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.

[13] F.G. Pagan. *Partial Computation and the Construction of Language Processors.* Prentice-Hall, 1990.