Towards Bridging the Gap Between Programming Languages and Partial Evaluation

A.-F. Le Meur, J.L. Lawall, Charles Consel

Towards Bridging the Gap Between Programming Languages and Partial Evaluation

Anne-Françoise Le Meur
INRIA/LaBRI
ENSEIRB, 1 avenue du
docteur Albert Schweitzer
33402 Talence Cedex, France
lemeur@labri.fr

Julia L. Lawall
Dept. of Computer Science
University of Copenhagen,
Universitetsparken 1
DK-2100 Copenhagen,
Denmark
julia@diku.dk

Charles Consel
INRIA/LaBRI
ENSEIRB, 1 avenue du
docteur Albert Schweitzer
33402 Talence Cedex, France
consel@labri.fr

ABSTRACT

Partial evaluation is a program-transformation technique that automatically specializes a program with respect to user-supplied invariants. Despite successful applications in areas such as graphics, operating systems, and software engineering, partial evaluators have yet to achieve widespread use. One reason is the difficulty of adequately describing specialization opportunities. Indeed, under-specialization or over-specialization often occurs, without any direct feedback to the user as to the source of the problem.

We have developed a high-level, module-based language allowing the programmer to guide the choice of both the code to specialize and the invariants to exploit during the specialization process. To ease the use of partial evaluation, the syntax of this language is similar to the declaration syntax of the target language of the partial evaluator. To provide feedback to the programmer, declarations are checked throughout the analyses performed by partial evaluation. The language has been successfully used by a signal-processing expert in the design of a specializable Forward Error Correction component.

1. INTRODUCTION

After having been intensively studied for the past twenty years, partial evaluation has now reached a mature state. Major advances have been made in understanding partial evaluation, both theoretically and practically. Many variations have been explored with respect to language paradigms and features. There are now partial evaluators for widely used languages such as C [1, 8] and Java [26].

Partial evaluation is essentially an aggressive form of constant propagation that specializes a program with respect to programmer-supplied invariants. A partial evaluator simplifies *static* computations, which depend only on information that can be inferred from the invariants and the program structure, and reconstructs the remaining *dynamic* computations to form the specialized program. Numerous strategies [5, 9, 12, 15] have been developed for the specialization process, leading to a wide variety of effects that can be achieved by partial evaluation; many of these techniques have been implemented in practical partial evaluators. These tools have been applied to a wide range of realistic applications in domains such as operating systems [17, 23], scientific computing [4, 18], graphics [2, 16] and software engineering [21, 28].

Despite these advances, partial evaluation still cannot be considered completely successful, because existing tools have been almost uniquely used by designers of partial evaluators. There are three major reasons for this situation.

Inappropriate Program Structuring. A major stumbling block in the application of partial evaluation to complex programs has been the problem of detecting program patterns that offer specialization opportunities. Decomposing an implementation as collection of units that each implements a single functionality simplifies reasoning about the program structure, and can thus highlight such program patterns. Nevertheless, even when a program is cleanly structured according to the needs of the implementation, this decomposition may be too coarse-grained to be the basis of an adequate specialization strategy, either because specialization should be applied to code spread across multiple units, or because some code in these units does not present significant specialization opportunities.

Complex Configuration of Partial Evaluators. In practical applications, advanced partial-evaluation features do not always justify their costs. Thus, it is desirable to allow the programmer to control how such features are used in the partial evaluation process. Furthermore, when partial evaluation is applied to only a fragment of a program, correct specialization requires that the programmer describe the interaction between the selected program fragment and the rest of the program. Providing this kind of configuration information can be complex and error-prone, and de-

PEPM '02, Jan. 14-15, 2002 Portland, OR, USA

tailed knowledge of the internals of the partial evaluator may be needed to understand the effect of particular declarations. When such declarations are provided in an unstructured way, the expertise incorporated in a successful use of specialization is not easily transferable.

Coarse-Grained Specialization Declarations. Successful specialization typically requires that the programmer have an intuitive understanding of how known information should propagate through the program. Nevertheless, partial evaluators typically do not provide adequate abstractions to describe specialization intentions. Indeed, to specialize a program, the programmer generally only provides information about the entry-point arguments and global variables. Partial evaluators give little high-level feedback as to the degree of specialization that can be achieved. This problem is compounded by the approximations that must necessarily be performed by any program analysis and transformation tool.

These shortcomings reflect a gap between the language used to write a program and the information needed to successfully exploit specialization opportunities. Currently, there exists insufficient support to help a programmer specify what specializations should be performed in a complex program.

Our Approach

To bridge the gap between programming languages and partial evaluation, we introduce a language that allows the programmer to declare *specialization scenarios* for a given program. A specialization scenario specifies *what* to specialize: which functions and data structures within the code are of interest for specialization, and what is their appropriate specialization context. Our approach complements existing partial evaluation techniques by providing the following:

A Language for Declaring Specialization Scenarios. We define a module-based language that allows the programmer to declare what code fragments and data structures should be processed by the specializer. The organization of this information into *specialization modules* localizes specialization information related to a particular functionality, and facilitates the understanding and reuse of specialization scenarios.

Automatic Partial Evaluator Configuration. Our language is independent of the configuration language of the targeted partial evaluator, and indeed borrows heavily from the standard C declaration syntax. Specialization modules can be translated automatically into the configuration declarations accepted by existing partial evaluators, modulo the features provided by the target partial evaluator.

Checking of Fine-grained Specialization Declarations. A specialization scenario declares the binding-time properties of each function, data structure and global variable. These declarations can be checked during the preliminary analyses performed by the partial evaluator. Such checking ensures that information derived from the programmer-supplied invariants is propagated according to the programmer's expectations, as defined by the declarations, thus improving the predictability of partial evaluation.

Contributions

We present a declaration language aimed at bridging the gap between the C programming language and existing partial evaluators. Our contributions can be summarized as follows.

- We make partial evaluation accessible to non-experts by providing a high-level language that is close to the C programming language and abstracts complex partial evaluation concepts into easy-to-use and intuitive declarations.
- The use of this language makes partial evaluation more predictable because it allows the programmer to specify how binding-time properties should be maintained throughout the program.
- We have developed a compiler for our language that automatically generates configuration declarations for the partial evaluator Tempo. Tempo has been developed by the Compose group [8] and successfully applied to applications in various domains [17, 21, 22, 23].
- As a proof of concept, we have implemented in Tempo our strategy to check the coherence between the programmer's declarations and the transformations performed during specialization.

We are currently using our approach in the context of a larger project in the Compose group to develop specializable system components together with operating-system programmers. Such programmers are typically unfamiliar with partial evaluation, but are very knowledgeable and demanding regarding code optimization. We have already developed, with the help of non-experts in partial evaluation, generic components that enable the rapid generation of efficient Forward Error Correction encoders through specialization [19]. We have found that the use of specialization scenarios makes partial evaluation more accessible and effective, makes the expertise needed to specialize an application explicit and re-usable, and promotes the development of generic code without sacrificing performance.

The rest of this paper is organized as follows. First, we define the declaration language in Section 2. Then, Section 3 describes the compilation of the specialization declarations. Section 4 first describes our strategy to ensure that the desired degree of specialization is achieved and then presents the specialization process and its correctness. Related work is discussed in Section 5. Finally, Section 6 concludes and suggests future work.

2. DECLARATION LANGUAGE

Applying specialization to an already-developed program is a difficult task. The programmer has to study the code to identify code fragments that contain interesting specialization opportunities, and then to describe the interaction between these code fragments and the rest of the program. A promising alternative is to take specializability into account during program development. At this stage genericness can be encoded using strategies that are known to specialize well, and the program can be structured such that there is

a clean separation between the code to be specialized and the rest of the program. So that this design effort can usefully guide specialization, we propose a language that allows the programmer to clearly describe the scenarios in which specialization is beneficial.

2.1 Design Decisions

At minimum, description of a specialization scenario must declare the program point at which specialization should begin and the variables with respect to which the code should be specialized. Nevertheless, experience has shown that this amount of information is not sufficient to ensure that an automatic partial evaluator can carry out the programmer's intentions. Declaring the binding-time properties of every program construct, however, is excessively burdensome for the programmer and can over-constrain the specialization process. Instead, we propose that the programmer declare the binding-time properties of global variables, data-structure components, and function parameters. The binding times of these constructs are then fixed according to these declarations at each reference point throughout the program, but the binding times of local variables, for which the programmer does not provide any declarations, can vary according to the strategy taken by the partial evaluator. This approach allows checking of the programmer's intentions pervasively throughout the program, but allows the specialization process to benefit from particular features of the partial evaluator intraprocedurally.

To reduce the overhead in learning how to apply specialization, the means of specifying a specialization scenario should borrow as much as possible from the syntax of the target language. Our specification language is targeted towards C programs, and thus the declarations of specialization properties amount to annotated C declarations. To facilitate understanding of a specialization process, related scenarios are grouped into modules. Multiple modules can be declared for a single code fragment, corresponding to multiple visions of its specializability. Alternatively, multiple scenarios can be declared for a given C construct in a single module.

2.2 Example

The function `dot` shown below, implements the multiplication of two integer vectors. It first checks that the vectors have the same size, aborting using the function `error` if they do not, and then computes the dot product.

```
struct vec {int* data; int length;};
int dot(struct vec* u, struct vec* v) {
  int i = 0;
  int sum = 0;
  if (u->length != v->length) error();
  while(i < u->length) {
    sum += u->data[i] * v->data[i];
    i++;}
  return sum;
}
```

The function `dot` is an example of a function that presents several specialization opportunities. If the size of both vectors is static, the size test can be reduced and the while loop can be unrolled. If the data of one of the vectors is static, then this data can be inlined into the code. The specialization module `vector`, shown in Figure 1, describes these specialization opportunities. The opportunities associated with each program construct are as follows:

```
Module vector {
 ...
 Defines {
  From dotproduct.c {
   VecDS::struct vec                                  (1)
          {D(int*) data; S(int) length;};
   VecSS::struct vec                                  (2)
          {S(int*) data; S(int) length;};
   Btdot1::intern dot                                 (3)
           (VecDS(struct vec) S(*) u,
            VecDS(struct vec) S(*) v)
           needs{Bterr;};                             (4)
   Bterr::extern error();                             (5)
   Btdot2::intern dot                                 (6)
           (VecSS(struct vec) S(*) u,
            VecDS(struct vec) S(*) v)
           needs{Bterr;};
 ...}...}
 Exports {VecDS; VecSS; Btdot1; Btdot2; ...}          (7)
}
```

Figure 1: Specialization module `vector`

Data structure `vec`: The scenarios `VecDS` (line 1) and `VecSS` (line 2) describe binding-time properties of the structure `vec` that can allow useful specialization. In both scenarios, the variable `length`, representing the size of the vector, is declared to have the type `S(int)`, indicating that the size should be static, and thus enabling array-bounds checks to be eliminated by specialization. In `VecDS`, the variable `data`, representing the data of the vector, is declared to have the type `D(int *)`, indicating that the data should be dynamic. In `VecSS`, the data is required to be static, which allows this data to be inlined.

Function `dot`: The scenarios `Btdot1` and `Btdot2` specify several scenarios for the `dot` function. The scenario `Btdot1` (line 3) says that `dot` can be specialized when the pointers `u` and `v` are both static and when the vectors to which they refer satisfy the scenario `VecDS`. The scenario `Btdot1` must also describe the specialization behavior of other functions referenced by `dot`, here only `error`. The declaration `needs{Bterr;}` (line 4) indicates that the scenario `Bterr` (line 5) should be used whenever `error` is invoked. This scenario indicates that `error` should be considered as `extern`, meaning that it is of no interest for specialization. The specialization module also defines the scenario `Btdot2` (line 6), which specifies that the function can be specialized if the size of both vectors is static, and the data of the vector referenced by `u` is static as well. A third scenario could be defined for the case where the data of the vector referenced by `v` is static.

Once all scenarios have been defined, we make them accessible to other specialization modules by adding the scenario names to the section `Exports` (line 7). The scenario `Bterr` is not exported, as the function `error` is for local use only.

```
module     ::= Module module_id
               {(imports)? defines exports}
imports    ::= Imports
               {(From file {(scen_id;)+})* }
defines    ::= Defines
               {(From file {(definition;)+})* }
definition ::= global_def | struct_def | proc_def
global_def ::= scen_id::bt_info var_id
struct_def ::= scen_id::struct struct_id
                        {(bt_info field_id;)+ }
proc_def   ::= scen_id::def_mode proc_id((params)?)
                        (needs_list)?
params     ::= (bt_info p_id,)* bt_info p_id
             | (bt_info,)* bt_info
def_mode   ::= extern | intern
needs_list ::= needs {(scen_id;)+ }
bt_info    ::= base_bt (base_type)
             | pointer | array | struct
base_bt    ::= S | D
pointer    ::= bt_info base_bt(*)
array      ::= bt_info base_bt([])
struct     ::= scen_id(struct struct_id)
base_type  ::= int | long | char | ...
exports    ::= Exports {(scen_id;)+ }
```

Figure 2: Syntax of the specification language

This simple example shows how, using our high level language, the programmer can describe multiple scenarios in which code fragments specialize well.

2.3 Specification Language

The syntax of the language of specialization declarations is defined in Figure 2. A specialization module is introduced by the keyword `Module` and consists of the name of the module, followed by three sections: *imports* (which is optional), *defines* and *exports*.

Imports The imports section, introduced by the keyword `Imports`, allows the current module to refer to specialization scenarios defined in other modules. Scenarios can be imported from multiple modules, each specified by the declaration

`From` *file* `{`(*scen_id*`;`)$^+$`}`

where *file* refers to a file containing the definition of another module. Within each such declaration, multiple scenarios can be imported.

Defines The defines section, introduced by the keyword `Defines`, lists a collection of specialization scenario definitions. Each scenario is associated to a C construct defined in the source file specified by the declaration `From` *file*. Scenarios can be declared for the following kinds of program constructs:

Global variables The declaration

scen_id`::`*bt_info var_id*

creates a specialization scenario *scen_id*, that associates the specialization information *bt_info* with the global variable *var_id*. Specialization information is described by decorating the type of the variable with either simple binding times (`S` or `D`) or, in the case of a structure type, with the name of a specialization scenario. These declarations must be well-formed: a dynamic pointer cannot be declared to point to a static value.

Data structures The declaration

scen_id`::struct` *struct_id* `{`(*bt_info field_id*`;`)$^+$`}`

creates a specialization scenario *scen_id* that describes the specialization information associated with each field of the structure *struct_id*.

Functions The declaration

scen_id`::`*def_mode proc_id* `((`*params*$)^?$`)`(*needs_list*$)^?$

creates a specialization scenario *scen_id* that describes the specialization information of the parameters of the function *proc_id*. The definition mode *def_mode* of this function scenario is `intern` if the function should be specialized, and `extern` otherwise. If the function *proc_id* calls other functions or declares local variables of structure type, the specialization scenarios that apply to these objects must be declared as well, using the *needs_list*.

Exports The exports section, introduced by the keyword `Exports`, lists the scenarios defined by the current module that are exported for use in other modules.

Specialization modules can be easily constructed by copying and annotating declarations already present in the C source file. Nevertheless, one could imagine an interactive tool that allows the user to decorate the source program with binding times, and that then automatically generates a specialization module expressed using the above syntax. We leave this development to future work.

3. SPECIALIZATION MODULE COMPILATION

Specialization modules can be compiled into the declarations processed by a particular partial evaluator. The inputs to such a compiler are the name of a scenario and the name of the module that exports this scenario. Based on this information, the compiler collects a graph of dependent scenarios, in which it records for each scenario the name of the corresponding C construct, the source file defining the construct, the binding time constraints associated with the construct, and the referenced scenarios. For each specialization module associated with a scenario in this graph, the compiler checks coherence within the module (*e.g.,* only scenarios defined in the current module can be exported) and between the module and the source program (*e.g.,* the signature of a construct declared in a module must match the signature of the corresponding construct in the source program indicated by the `From` keyword). The information stored in the graph of dependent scenarios is then used both to configure the

targeted partial evaluator and to provide the binding-time constraints to check during the analyses.

We have implemented a compiler that processes a specialization module and a source program to produce the configuration information required by the partial evaluator Tempo. Because Tempo expects the source program to be presented as a single source file, the compiler extracts the fragments of code to be specialized from the source program and reassembles them to form a single C source file. Tempo configuration directives are also automatically generated. For example, a description of the binding time of each entry-point argument is constructed in the format expected by Tempo. When some arguments have complex types, writing such a description by hand can be tedious and error-prone.

4. VERIFICATION AND SPECIALIZATION

A specialization module declares the context to which a code fragment should be specialized, and how specialized functions referred to within the code fragment should interact. To ensure that this degree of specialization is achieved, we augment the standard partial evaluation process to check that these declarations are respected. The key issue is to ensure that information declared to be static is propagated pervasively through the program.

Offline partial evaluation is divided into an analysis phase that determines what can be simplified based on the known information, followed by a specialization phase that simplifies the selected computations and reconstructs the remaining computations to form the specialized program. Declarations are checked during the analysis phase. We begin by describing the strategies used for these checks. We then show that specialization, which is defined independently of the binding-time declarations, both respects the semantics of the source program (the standard correctness criterion for a partial evaluator) and respects the declarations, thus ensuring that the desired degree of specialization is achieved.

4.1 Verification Strategy

The degree of specialization is determined by the binding times of location references, because these are the points through which static values are propagated within the program. Thus, the analysis of each variable reference and dereference expression checks that the binding time matches the declarations on the possible referenced locations. In other cases, we follow a more lazy verification strategy, allowing the inferred binding time for a location to conflict with the specified binding time, at points where the value of the location is not actually used.

Several aspects of our verification strategy are illustrated by the following artificial example, where all irrelevant computations have been removed. For conciseness, we indicate binding-time constraints by direct annotation of the types in the source program, rather than in a separate specialization module. Possible annotations are $\mathcal{S}$ (static), $\mathcal{D}$ (dynamic), and $\mathcal{U}$ (unspecified). The aliases of a dereference expression are indicated by a superscripted set.[1] Static terms are underlined.

```
int^D d;

void main() {
  int^U a = 3;
  f (&a, &a);                                   (1)
}

int f(int^D *^S x, int^S *^S y) {
  int^U *^U *^U z = &y;
  y = &d;                                       (2)
  *x^{a} = d;                                   (3)
  if (d) x = NULL;                              (4)
  x = y;                                        (5)
  return *(*z^{y})^{d};                         (6)
}
```

Binding times of locations are modified or referenced in the analysis of function calls, assignments, conditionals, variables references, and dereference expressions. The treatment of these constructs interacts with the verification process as follows:

Function calls: The specialization scenario for a function specifies binding times for the parameters at each possible level of indirection. All of these binding times are checked at the call site. In the example above, the scenario for `f` specifies that both arguments should be static pointers, and that a dereference of `x` should be dynamic while a dereference of `y` should be static. At the call site, in line 1, both arguments are `&a`, which is a static pointer to a static value.[2] Because a static value can be coerced to be dynamic, a completely static integer-pointer binding time of `&a` is compatible with the declaration of `x` as $\texttt{int}^{\mathcal{D}}\ *^{\mathcal{S}}$ and the declaration of `y` as $\texttt{int}^{\mathcal{S}}\ *^{\mathcal{S}}$.

Assignments: For an assignment, we adopt a more lazy strategy. We check that the binding time of the assigned value is compatible with the constraints on the possible affected locations, to verify that information is flowing through the program according to the programmer's intentions, but do not check the compatibility of constraints and binding times for aliases implicitly affected by the assignment.

Verification of the assignment `y = &d` (line 2) illustrates this laziness. The static binding time of `&d` satisfies the declaration of `y` as $\texttt{int}^{\mathcal{S}}\ *^{\mathcal{S}}$, despite the fact that `d` is declared to be dynamic. The correspondence between the binding time of `d` and the specification on the result of dereferencing `y` is not checked at this point, because the binding time of `*y` has no impact on specialization of the assignment statement.

Another example of the lazy treatment of assignments is provided by the assignment `*x = d` (line 3). The assignment itself satisfies the declaration for the affected locations `*x` and `a`, but does not satisfy the declaration for `*y`, which also has `a` as an alias. Again, this mismatch is not detected until there is a reference to `*y`.

[1] Some form of alias analysis within the partial evaluator is essential for correct treatment of a language including pointers, such as C. Alias information is not derived from the declarations in the specialization module.

[2] The address of a variable is always considered static.

Dynamic conditionals: A standard technique for treating a static assignment in a dynamic conditional or loop is to consider the possibly affected locations to be dynamic following the conditional or loop construct. We do not check this inferred binding time until such a location is actually referenced.

In line 4, `x` is assigned to `NULL` within a dynamic conditional statement. `NULL` is static, in correspondence to the constraint on `x`, but following the conditional statement, `x` is considered dynamic. This adjustment of the binding time is not checked. The explicit assignment of `x` to `y` in line 5 restores the correct binding time of `x`.

Variable reference or dereference expression: In the case of a variable reference or a dereference expression, we check that the constraint on the location is compatible with its inferred binding time, but do not check such compatibility for all possible dereferences of the location. This laziness is illustrated by the reference to `y` in line 5. Although the previous assignment to `y`, `y = &d`, causes the binding time of `*y` to conflict with the declaration on `y`, this declaration is not checked in line 5, which only references the value of `y` itself.

A dereference that involves an alias created by function-parameter bindings must respect both the declaration for the referenced location and the declarations for the parameters. The dereference `**z` in line 6 respects the constraints on `z` and the referenced location `d`, both of which allow the value to be dynamic, but because the relationship between `z` and `d` is derived from the parameter `y`, which requires that the dereferenced value should be static, an error is signaled.

We show the feasibility of this approach by defining an offline partial evaluator for a simple imperative language in Appendix A. This partial evaluator is flow sensitive and allows for a precise treatment of pointers via an alias analysis. Loops and recursion are not treated, but can be added using standard techniques [3], as done in our implementation.

4.2 Specialization

The specialization phase simplifies the static constructs and reconstructs the dynamic constructs to form the specialized program. Because programmer declarations are completely encoded into the results of the analyses, they are not directly referenced during the specialization phase. A standard specializer can thus be used.

Soundness with Respect to the Semantics: The standard criterion for soundness of a partial evaluator is that, if specialization with respect to some static inputs succeeds, execution of the specialized program with respect to some dynamic inputs should produce the same result as execution of the original program with respect to both the static and the dynamic inputs. We can show that this property holds, independently of the constraints, whenever at each point in the analysis process the binding-time analysis annotates each location reference with a binding time that is *greater than or equal to* the current inferred binding time for the location, as recorded in the current binding-time environment. Because the use of constraints in each case produces a binding time with this property, soundness of the partial evaluator is not affected by the verification of constraints during the binding-time analysis.

Soundness with Respect to the Programmer's Declarations: Intuitively, specialization that respects the programmer's declarations should produce a specialized program in which all variables declared to be static have been removed. Unfortunately, as outlined in Appendix A, the existence of non-liftable values (static values, such as addresses, that have no meaningful representation in the specialized program) implies that this goal is not achievable without undesirably restricting the set of programs that is accepted for specialization. We thus allow variables that are declared to be static but have a non-liftable value to occur in certain contexts in the specialized program.

We distinguish between two kinds of contexts in which an expression can occur: *static contexts* and *dynamic contexts*. A static context is one where simplification can necessarily occur if the context is filled with a static expression. For example, a context whose hole is the test expression of a conditional statement is static. Conversely, a dynamic context is one where no simplification can necessarily occur even if the context is filled with a static expression. For example, a context whose hole is one argument of a binary operator is dynamic.

We annotate the semantics of the source language such that each step in the treatment of an expression is annotated with the context in which the expression occurs. For example, the annotated semantics of a variable reference and a dereference expression are as follows, where b is either S or D according to whether the context of the current expression E is static or dynamic respectively, ρ is a store mapping locations to values, and $\mathtt{x}_\ell$ is the location associated with the variable `x` (this location is determined implicitly):

$$\frac{b, \mathtt{x}_\ell \mapsto v \in \rho}{b, \rho \models_{\mathrm{e}} \mathtt{x} : v} \qquad \frac{\mathsf{D}, \rho \models_{\mathrm{e}} E : l \qquad b, l \mapsto v \in \rho}{b, \rho \models_{\mathrm{e}} *E : v}$$

The annotations have no effect on the values associated with expressions or locations by the semantics. Thus, $b, \rho \models_{\mathrm{e}} E : v$ in the annotated semantics if and only if $\rho \models_{\mathrm{e}} E : v$ in the original semantics, and $b, l \mapsto v \in \rho$ in the annotated semantics if and only if $l \mapsto v \in \rho$ in the original semantics.

Using this annotated semantics, we can then prove that execution of the specialized program according to the annotated semantics never references a location associated with a variable declared to be static in a static context, and never references a location associated with a variable of liftable type (non-pointer type, for C programs) declared to be static in a dynamic context. The proof follows from the fact that the evaluation-time analysis only reannotates a static expression having a non-liftable value when this expression occurs in a dynamic context.

5. RELATED WORK

The difficulty of obtaining a desired specialized program by automatic techniques alone has led to a variety of approaches that allow the programmer to control the specialization process. Most of the previous proposals require annotations to be placed in the source program, in some cases violating the

syntax of the original source language. Some strategies allow only one possible annotation per function definition. Except for specialization classes [29], none of the previous strategies provide any structuring of the specialization declarations.

DyC The DyC run-time specialization system includes annotations that allow the programmer to control various aspects of the specialization process, including the propagation of specialized values [13]. Annotations are distributed throughout the source code, implying that the source program is no longer a standard C program.

Based on user annotations identifying static variables, DyC automatically infers the region of code to be specialized. Although the annotation language allows the programmer to control many aspects of the strategy used in this inference, no feedback is given as to the region actually selected. Besides the lack of precision inherent in any static analysis engine, the inference of the specialized region in DyC is further complicated by the integration of DyC with the Multiflow compiler, which has been observed to obscure the relationship between the programmer annotations and the source code [13].

C-Mix The C-Mix partial evaluator for the C programming language provides annotations that control the binding-times of variables and external function calls [6]. Annotations can be provided in the source file, in a script file, or on the command line. Variables can be annotated as either static or dynamic. If a dynamic binding time is inferred for a variable declared to be static, an error occurs. If a static binding time is inferred for a variable declared to be dynamic, the binding time is coerced to be dynamic. While we use the same verification strategy, the flow-insensitivity of C-Mix implies that some programs that are accepted because of the laziness of our verification strategy are rejected by C-Mix. Overall, the annotations of C-Mix are directed towards variables, whereas our annotations are more fine-grained, applying to each kind object that can be referenced via a variable, as indicated by the variable's type. For example, C-Mix annotations cannot declare the binding time of a variable dereference to be different from the binding time of the variable itself.

Schism The Schism partial evaluator for a pure subset of the Scheme programming language provides the *filter* mechanism that allows the programmer to specify which static arguments should be propagated into the body of a specialized function [7]. Because filters use the normal Scheme syntax, they can be macro-expanded away to allow normal execution of the source program. A filter can include computation on the binding-times of the arguments, thus allowing the programmer to declare multiple strategies for a single function. Nevertheless, specialization information is still distributed across the program source code.

Fabius The Fabius run-time code generation system for the ML programming language requires the user to separate the static and dynamic arguments of each function by currying, such that the first argument contains the static information and the second argument contains the dynamic information [20]. This strategy typically requires modification of the source program and can express only one specialization strategy per function, but because the binding-time specification uses no special syntax, it allows the program to be executed normally as well as being specialized.

Multi-level languages Multi-level languages, such as MetaML [27] and ‘C [11], provide a high-level notation for describing the construction of code fragments. While such languages allow the construction of specialized code satisfying precise programmer specifications, such complete annotation is more tedious than the annotation of function parameters suggested by our approach. Furthermore, the source program is not written using a standard language, and can thus only be processed by the specific tool.

Specialization classes Specialization classes are a declarative notation for specifying specialization opportunities in Java programs [29]. Declarations focus on the interaction between the original program and the specialized code. A specialization class indicates the names of the methods at which to begin specialization, identifies the static arguments, and selects between compile-time and run-time specialization. This information is compiled into a description of the specialization context of the entry point, and the original definition of the entry point function is modified to choose between the specialized and unspecialized versions. Nevertheless, there is no programmer control of how the specialization process is carried out.

Conceptually, our approach is complementary to the use of specialization classes. A specialization module provides user control over the specialization process itself. Like a specialization class, a specialization module allows the programmer to describe the binding-time properties of the entry point. The declarations of the binding times of function parameters, global variables, and data structures allow the programmer to further describe how the information derived from the static entry-point arguments should propagate through the program.

Annotations used in other kinds of automated tools The importance of providing user information to direct and make tractable automatic techniques has long been recognized in the areas such as theorem proving [10], and is beginning to be recognized in the area of program analysis as well [14, 25].

6. CONCLUSION

We have presented a high-level language that enables the programmer to declare what code fragments and data structures of a program should be specialized. Our language hides complex partial evaluation concepts and allows non-experts to intuitively declare specialization properties. These properties are checked during the specialization process. Specialization declarations are organized into specialization modules, allowing a structured and modular approach to specialization. This approach has been implemented in the partial evaluator Tempo.

Our principal goal has been to enhance the usability of partial evaluators. To this end, we have designed a declaration language that is independent of particular features of a specific partial evaluator. The verification of these declarations during partial evaluation improves the predictabil-

ity of the specialization process. Finally, the separation of the specialization declarations from the source program and the organization of related specialization declarations into specialization modules facilitates the reusability of acquired specialization expertise.

Our language permits the programmer to describe how static information should propagate through the program, but not to describe what transformations should be performed based on this information. For example, to limit code size, it can be useful to avoid unrolling loops. We have already begun to extend our declaration language to allow the programmer to specify constraints on the values of function parameters and global variables. This feature enables a better control of constant propagation and thus provides a means to disable loop unrolling.

7. ACKNOWLEDGMENTS

The first and third authors were partially supported by the ITEA project ESAPS. The second author was partially supported by a grant from the Danish Natural Science Research Council.

8. REFERENCES

[1] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[2] P. Andersen. Partial evaluation applied to ray tracing. In W. Mackens and S. Rump, editors, *Software Engineering in Scientific Computing*, pages 78–85. Vieweg, 1996. DIKU Technical Report D-289.

[3] J. M. Ashley and C. Consel. Fixpoint computation for polyvariant static analyses of higher-order applicative programs. *ACM Transactions on Programming Languages and Systems*, 16(5):1431–1448, 1994.

[4] A. Berlin and R. Surati. Partial evaluation for scientific computing: The supercomputer toolkit experience. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 133–141, Orlando, FL, USA, June 1994. Technical Report 94/9, University of Melbourne, Australia.

[5] A. Bondorf. Improving binding times without explicit CPS-conversion. In *ACM Conference on Lisp and Functional Programming*, pages 1–10, San Francisco, CA, USA, June 1992. ACM Press.

[6] C-Mix$_{/\mathrm{II}}$ user and reference manual. http://www.diku.dk/research-groups/topps/activities/cmix/download/, 2000.

[7] C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.

[8] C. Consel, L. Hornof, F. Noël, J. Noyé, and E. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, Feb. 1996.

[9] C. Consel and S. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.

[10] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

[11] D. Engler, W. Hsieh, and M. Kaashoek. ‘C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 131–144, St. Petersburg Beach, FL, USA, Jan. 1996. ACM Press.

[12] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. The benefits and costs of DyC's run-time optimizations. *ACM Transactions on Programming Languages and Systems*, 22(5):932–972, 2000.

[13] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[14] B. Grobauer. Cost recurrences for DML programs. In *ICFP 2001: International Conference on Functional Programming*, pages 253–264, Florence, Italy, Sept. 2001. ACM Press.

[15] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *Theoretical Computer Science*, 248(1–2):3–27, 2000.

[16] T. Knoblock and E. Ruf. Data specialization. In PLDI'96 [24], pages 215–225. Also TR MSR-TR-96-04, Microsoft Research, February 1996.

[17] J. Kono and T. Masuda. Efficient RMI: Dynamic specialization of object serialization. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, pages 308–315, Taipei, Taiwan, Apr. 2000. IEEE Computer Society Press.

[18] J. Lawall. Faster Fourier transforms via automatic program specialization. In J. Hatcliff, T. Mogensen, and P. Thiemann, editors, *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, volume 1706 of *Lecture Notes in Computer Science*, pages 338–355, Copenhagen, Denmark, 1999. Springer-Verlag.

[19] A.-F. Le Meur, C. Consel, and B. Escrig. Guaranteed configurability of components via specialization modules. Research Report 1256-01, LaBRI, Bordeaux, France, Mar. 2001.

[20] P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI'96 [24], pages 137–148.

[21] R. Marlet, S. Thibault, and C. Consel. Efficient implementations of software architectures via partial evaluation. *Journal of Automated Software Engineering*, 6(4):411–440, Oct. 1999.

[22] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, C. Goel, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19:217–251, May 2001.

[23] G. Muller, R. Marlet, E. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[24] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, USA, May 1996. ACM SIGPLAN Notices, 31(5).

[25] B. Ryder. A position paper on compile-time program analysis. *ACM SIGPLAN Notices*, 32(1):110–114, Jan. 1997.

[26] U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390, Lisbon, Portugal, June 1999.

[27] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, Amsterdam, The Netherlands, June 1997. ACM Press.

[28] S. Thibault and C. Consel. A framework for application generator design. In *Proceedings of the Symposium on Software Reusability*, Boston, MA, USA, May 1997.

[29] E. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, GA, USA, Oct. 1997. ACM Press.

APPENDIX

A. ANALYSES

We define a specializer that relies on three analyses: alias analysis, binding-time analysis, and evaluation-time analysis. Alias analysis associates each dereferenced expression $*E$ with the set of *abstract locations* (program variables or dereferences of program variables) to which it can refer. Any analysis algorithm can be used. No verification is performed during alias analysis, but alias information is crucial to be able to transmit binding-time and constraint information across pointers. Binding-time analysis classifies each expression as either static, indicating that its value depends only on information known during partial evaluation, or dynamic, indicating that its value depends on information not available until execution time. This phase checks that each location declared as static is inferred to be static, and coerces each location declared as dynamic to be dynamic. Evaluation-time analysis ensures the consistency of the specialized program. This phase guarantees that each variable referenced in the specialized program is also initialized in the specialized program, and addresses the problem of specializing a static expression for which the specialization-time value is not meaningful at run time.

We now present the source language and the analyses that affect the binding-time annotations, *i.e.,* the binding-time analysis and the evaluation-time analysis, in more detail.

Source Language

The source language is a simple, non-recursive, imperative language including one-argument functions, assignment statements, conditional statements, and dereference expressions. A program consists of a collection of global variables and functions, defined as follows:

$$
\begin{array}{rrrl}
G & \in & \text{Global} & ::= T\ \texttt{x} \\
F & \in & \text{Function} & ::= \texttt{f(}T\ \texttt{x)}\,S \\
T & \in & \text{Type} & ::= \texttt{int} \mid T\,\texttt{*} \\
S & \in & \text{Statement} & ::= L\texttt{ = }E\texttt{;} \mid \texttt{\{}T\ \texttt{x;}\ S\texttt{\}} \mid \texttt{\{}S_1\ S_2\texttt{\}} \\
 & & & \quad \mid \texttt{if (}E\texttt{) then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{f(}E\texttt{)} \\
E & \in & \text{Expression} & ::= \texttt{x} \mid \texttt{\&x} \mid *E \\
L & \in & \text{L-expression} & ::= \texttt{x} \mid *E
\end{array}
$$

The entry point is assumed to be a statement invoking one of the defined procedures. The semantics is standard.

Binding-Time Analysis

Binding-time analysis infers a binding time for each program construct based on the inferred binding time for each abstract location and on the programmer's declarations. Binding times are denoted as S (static) and D (dynamic), where $\mathsf{S} \sqsubseteq \mathsf{D}$. Programmer declarations are denoted as $\mathcal{S}$, indicating that the location must be considered static when referenced, $\mathcal{D}$, indicating that the location must be considered dynamic when referenced, and $\mathcal{U}$, indicating that there is no constraint on the binding time of the location. These constraints are ordered as $\mathcal{U} \sqsubseteq \mathcal{S}$ and $\mathcal{U} \sqsubseteq \mathcal{D}$. The operation $c \oplus b$ produces a binding time compatible with both the constraint c and the binding time b, and is defined as follows:

$$\mathcal{S} \oplus \mathsf{S} = \mathsf{S} \qquad \mathcal{D} \oplus b = \mathsf{D} \qquad \mathcal{U} \oplus b = b$$

Note that $\mathcal{S} \oplus \mathsf{D}$ is not defined, because $\mathcal{S}$ and D are incompatible. The operation $\oplus$ can thus be used both to compute a new binding time and to check compatibility.

Let Σ be an environment mapping each location to a constraint, Γ be an environment mapping each location to a binding time, and Φ be an environment mapping each function name to its definition. Correct annotation of a statement S is specified by the judgment $\Sigma, \Gamma, \Phi \vdash_{\mathrm{s}} S : \hat{S}, \Gamma'$, where $\hat{S}$ is the binding-time annotated counterpart of the statement S, and Γ' is a new binding-time environment re-

flecting the assignments processed while analyzing S. Correct annotation of an expression E is specified by the judgment $\Sigma, \Gamma \vdash_{\mathrm{e}} E : \hat{E}^b$, where b is a binding-time and $\hat{E}^b$ is a binding-time annotated expression. Finally, correct annotation of an L-expression L is specified by the judgment $\Sigma, \Gamma \vdash_{\mathrm{l}} L : \hat{L}^b$, where b is a binding-time and $\hat{L}^b$ is a binding-time annotated L-expression. Binding times that satisfy these rules can be inferred by standard techniques.

Most of the well-formedness rules are standard for a flow-sensitive binding-time analysis of an imperative language [15]. We thus present only the rules that involve the constraints derived from the specialization module:

Function Call: A specialization scenario for a function declares the expected binding time of the parameter, as well as the expected binding time of all possible dereferences of the parameter, according to its type. The well-annotatedness rule for a function call is as follows:

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash_{\mathrm{e}} E : \hat{E}^b \\ \\ \mathtt{f} \mapsto (\mathtt{x}, [c, c_1, \ldots, c_m], S) \in \Phi \\ \mathcal{L}^*(E) = [(\kappa_1, \delta_1), \ldots, (\kappa_m, \delta_m)] \\ (c_1 \sqcup (\bigsqcup\{\Sigma(l) \mid l \in \kappa_1\}) \sqcup (\bigsqcup\{\Sigma(p) \mid p \in \delta_1\})) \oplus \\ \quad (\bigsqcup\{\Gamma(l) \mid l \in \kappa_1\}) \\ \vdots \\ (c_m \sqcup (\bigsqcup\{\Sigma(l) \mid l \in \kappa_m\}) \sqcup (\bigsqcup\{\Sigma(p) \mid p \in \delta_m\})) \oplus \\ \quad (\bigsqcup\{\Gamma(l) \mid l \in \kappa_m\}) \\ \\ \Sigma' = \Sigma[\mathtt{x} \mapsto c, *^1\mathtt{x} \mapsto c_1, \ldots, *^m\mathtt{x} \mapsto c_m] \\ \Sigma', \Gamma[\mathtt{x} \mapsto c \oplus b], \Phi \vdash_{\mathrm{s}} S : S', \Gamma'[\mathtt{x} \mapsto b'_{\mathtt{x}}] \\ \mathsf{update\_fn}(\mathtt{f}, S')\end{array}}{\Sigma, \Gamma, \Phi \vdash_{\mathrm{s}} \mathtt{f}(E) : \mathtt{f}(\hat{E}^b), \Gamma'}$$

The analysis of a function call includes three parts: analysis of the argument (the judgment $\Sigma, \Gamma \vdash_{\mathrm{e}} E : \hat{E}^b$), verification that the binding-time of the argument and all possible dereferences of the argument match the specification of the parameter (the middle group of hypotheses), and finally analysis of the body of the called function (the final three lines of the hypotheses).

The declaration for the parameter specifies a binding time for each of the m possible levels of indirection, via the list $[c, c_1, \ldots, c_m]$ stored in the function environment Φ. The operation $\mathcal{L}^*(E)$ accesses the aliases of E at each possible level of indirection. At each level, there are two kinds of aliases: concrete aliases κ and dummy aliases δ. Concrete aliases are program variables, and are associated with both a binding time and a constraint. Dummy aliases are parameter dereferences. These locations only serve to propagate the constraints on the parameters into the analysis of the body of the function. At each level, the constraint on the parameter is checked to be compatible with the constraints on both kinds of aliases, and with the binding times of the concrete aliases.

The body S of the called function is analyzed with respect to the current constraint environment Σ extended with the constraints on the possible dereferences of the parameter ($*^i$ refers to i dereferences of $\mathtt{x}$), and the current binding-time environment Γ extended with the parameter bound to its binding time. The operator update_fn is then used to update an implicit store of annotated function definitions with the annotated body $\hat{S}$. The resulting binding-time environment Γ' reflects the side-effects made by the body of the called function to non-local variables.

Assignment Statement: The binding time of the assignment statement is the least upper bound of the binding times inferred for the subexpressions. This binding time is checked to be compatible with the constraints on both the concrete and dummy aliases. If there is only one concrete alias the new binding time of this location is the binding time of the assignment, as shown by the following rule:

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash_{\mathrm{l}} L : \hat{L}^{b'} \qquad \Sigma, \Gamma \vdash_{\mathrm{e}} E : \hat{E}^{b''} \qquad b = b' \sqcup b'' \\ \mathcal{L}(L) = (\{l\}, \delta) \qquad \Sigma(l) \oplus b \qquad \forall p \in \delta, \Sigma(p) \oplus b\end{array}}{\Sigma, \Gamma, \Phi \vdash_{\mathrm{s}} L = E\,; : \hat{L}^{b'} =^b \hat{E}^{b''}\,;, \Gamma[l \mapsto b]}$$

If there are multiple concrete aliases, then the new binding time of each possible location is the least upper bound of the binding time of the assignment and the previous binding time of the location.

Variable Reference: The binding time of a variable reference $\mathtt{x}$ is computed from the constraint $\Sigma(\mathtt{x})$ and the current binding time $\Gamma(\mathtt{x})$, as follows:

$$\Sigma, \Gamma \vdash_{\mathrm{e}} \mathtt{x} : \mathtt{x}^{\Sigma(\mathtt{x}) \oplus \Gamma(\mathtt{x})}$$

Dereference Expression: The binding time of a dereference expression takes into account the binding time of the dereferenced expression E, as well as the binding times and constraints associated with the possible aliases.

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash_{\mathrm{e}} E : \hat{E}^b \qquad \mathcal{L}(*E) = (\kappa, \delta) \qquad b' = \bigsqcup\{\Gamma(l) \mid l \in \kappa\} \\ c_{\mathrm{c}} = \bigsqcup\{\Sigma(l) \mid l \in \kappa\} \qquad c_{\mathrm{d}} = \bigsqcup\{\Sigma(p) \mid p \in \delta\}\end{array}}{\Sigma, \Gamma \vdash_{\mathrm{e}} *E : (*\hat{E})^{(c_{\mathrm{c}} \sqcup c_{\mathrm{d}}) \oplus (b \sqcup b')}}$$

Evaluation-Time Analysis

The forward dependency analysis performed by binding-time analysis is not sufficient to guide the construction of a meaningful specialized program. Two kinds of problems can occur. First, if a variable is referenced when considered dynamic, the reference can appear in the specialized program and all possible reaching initializations of the variable must appear in the specialized program as well, even those that are static. Second, if the specialization-time value of a static expression cannot be "lifted", *i.e.,* converted to syntax, the expression must be considered dynamic when it occurs in a dynamic context.[3] Both of these adjustments require the backwards propagation of information, and are performed by evaluation-time analysis [15].

When the evaluation-time analysis determines that a static assignment must appear in the specialized program, the assignment is reannotated to be both static and dynamic. This reannotation does not interfere with the propagation

[3]For compile-time specialization, for example, an address is not liftable.

of static information through the program, and thus the binding-time constraints continue to be satisfied.

The reannotation of a static, non-liftable value occurring in a dynamic context as dynamic can constrain the propagation of static information during specialization. Consider the following example:

```
int^D *^S x;

if (x != NULL)
  ... *(x+1) ...
```

The declaration for `x` indicates that the result of the expression `*(x+1)` is dynamic. Thus, the static pointer-typed expression `(x+1)` occurs in a dynamic context, and must itself be considered dynamic. This reannotation in turn implies that `x` is considered dynamic, in contradiction to the declaration $*^{\mathcal{S}}$ in the type of `x`, and that the addition is not performed during specialization. Nevertheless, triggering an error at this point, and thus requiring `x` to be declared to be completely dynamic, would preclude simplification of the conditional test, which only depends on static information. Thus, the constraint on a variable having a non-liftable value is only guaranteed to be satisfied when the variable occurs in a static context. This property is ensured by the checks performed in the binding-time analysis and by the fact that the evaluation-time analysis does not reannotate such references.