A Pragmatic Approach to Declaring Specialization Scenarios

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

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

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

Abstract.

Partial evaluation is a program transformation that automatically specializes a program with respect to invariants. Despite successful application 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, underspecialization or overspecialization often occurs, without any direct feedback as to the source of the problem.

We have developed a high-level, module-based language allowing the program developer 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, declarations are checked during the analyses performed by partial evaluation. The language has been successfully used by a variety of users, including students having no previous experience with partial evaluation.

1. Introduction

After having been studied intensively for the past twenty years, 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 (Andersen, 1994; Consel et al., 1996) and Java (Schultz et al., 1999).

Partial evaluation is essentially an aggressive form of constant propagation that specializes a program with respect to 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. Typically, this process is divided into two phases: a *binding-time analysis* phase that identifies the static computations, followed by a *specialization* phase that constructs the specialized program. This basic approach has been extended with numerous advanced features, leading to a wide variety of effects that can be achieved (Bondorf, 1992; Consel and Khoo, 1993; Danvy, 1998; Grant et al., 2000a; Hughes, 1998); 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 (Kono and Masuda, 2000; Muller et al., 1998), scientific computing (Berlin and Surati, 1994; Lawall, 1998), graphics (Andersen, 1996; Knoblock and Ruf, 1996) and software engineering (Marlet et al., 1999; Thibault and Consel, 1997).

Despite these advances, partial evaluation still cannot be considered completely successful, because existing tools have been almost uniquely used by designers of partial evaluators. Based on our experience[1] applying partial evaluation in a variety of application areas and on the literature, we have identified three reasons for this situation:

Inappropriate Program Structuring. A major stumbling block in the application of partial evaluation to a complex program has been the problem of detecting program patterns that offer specialization opportunities (Jones, 1996; Mock et al., 2000; Muth et al., 2000; Schultz et al., 2000). Decomposing an implementation as a collection of units that each implements a single functionality simplifies reasoning about the program structure, and can highlight such program patterns. Nevertheless, even when a program is cleanly structured according to the needs of the implementation, the granularity of this decomposition may be inappropriate 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.

In the object-oriented world, an example of a well-recognized program structure that organizes the code in a manner that does not directly match the needs of specialization is the Visitor design pattern (Gamma et al., 1994). When using this design pattern, a visitor provides a collection of functionalities relevant to a particular kind of object. In general, it is likely that only a few of these functionalities provide specialization opportunities, implying that there is no need to expose the entire visitor to the specialization process. Furthermore, there may be interactions between objects of multiple concrete visitor

[1] Our experience is derived from using the partial evaluators Tempo and Schism for specializing a wide range of applications, and from teaching others how to use partial evaluation, both at a workshop on Tempo and in yearly graduate courses.

classes. In this case, specialization must consider code spread across many different files of the program.

Complex Configuration of Partial Evaluators. In practical applications, advanced partial-evaluation features may specialize the code in ways that are not desired (*e.g.*, excessive specialization can cause code explosion) (Blazy, 2000; Christensen et al., 2000; Grant et al., 2000b). Thus, it is desirable to allow the program developer to configure the partial evaluator to control how such features are used in the partial evaluation process. Providing this kind of configuration information can be complex and error-prone, and detailed 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 program developer 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 developer 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.

The granularity of programmer guidance of program analyses and transformations is a problem that is more general than partial evaluation. Ryder has observed that there is a general need for greater control over the results produced by program analyses (Ryder, 1997). Several researchers concerned with these shortcomings have developed systems that provide fine-grain annotations of program to finely control the transformation process (e.g., in the context of partial evaluation, Taha and Sheard (Taha and Sheard, 1997), Engler *et al.* (Engler et al., 1996)).

Overall, these shortcomings suggest that there currently exists insufficient support to help the program developer specify what specializations should be performed in a complex program.

Our Approach

To simplify the task of using partial evaluation, we introduce a language that allows the developer to declare *specialization scenarios* for a given program. A specialization scenario is a declaration that identifies a function, global variable or data structure that is of interest for specialization and declares the binding-time context in which specialization involving this construct should be carried out. Our approach thus complements existing partial evaluation techniques by providing the following:

A Language for Declaring Specialization Scenarios. We define a module-based language that allows the program developer to declare what code fragments and data structures should be processed by the specializer. Related scenarios are collected into *specialization modules*, which localize specialization information related to a particular functionality and facilitate 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 targeted partial evaluator.

Checking of Fine-grained Specialization Declarations. Specialization scenarios declare 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 program invariants is propagated according to the program developer's expectations, as defined by the declarations, thus improving the predictability of partial evaluation.

A specialization scenario is analogous to a type declaration, where types describe what information is known or unknown during specialization, rather than sets of concrete values. The manner in which these declarations are used is also similar to the use of type declarations in a language that provides type inference such as ML. While ML type inference is sufficient to determine the type of most programs based only on programmer-provided information about the types of data structure components, programmer annotation of each function with its type improves the readability of the code and facilitates the debugging of type errors. The declaration of a specialization scenario for each function, data structure, and global variable provides similar advantages.

ML programs and the associated type declarations are organized into modules, which facilitate overall program understanding, simplify the replugging of new implementations of existing functionalities, and allow separate analysis. Specialization modules provide the first two advantages, as *e.g.*, any implementation of a function can be used that satisfies the binding-time constraints declared in the specialization scenario. The declarations provided in specialization modules could furthermore allow separate binding-time analysis of individual functions (although relying on global alias information), similar to the intra-procedural binding-time analysis of DyC (Grant et al., 2000b). We have not explored this option, however, as our focus is on declaring specialization opportunities, rather than on reorganizing the basic analysis strategy of the underlying partial evaluator.

Our aim is to allow the program developer to declare where static information should be propagated within the code to be specialized. We do not, thus, consider the reintegration of specialized code into the original source program. This issue is addressed by the *specialization classes* of Volanschi *et al.* (Volanschi et al., 1997), which are complementary to our work.

Contributions

The contributions of this paper are as follows:

- We make partial evaluation accessible to non-experts by providing a high-level language, close to the C programming language, that 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 developer 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 (Consel et al., 1996) and successfully applied to applications in various domains (Kono and Masuda, 2000; Marlet et al., 1999; McNamee et al., 2001; Muller et al., 1998).

- As a proof of concept, we have implemented in Tempo one possible strategy to check the coherence between the program developer's declarations and the transformations performed during specialization.

- We use examples drawn from the partial evaluation literature to illustrate how the feedback provided by the checking of specialization declarations helps the developer identify and avoid problems that arise when applying partial evaluation,

- We have developed two graphical interfaces. One interface allows the program developer to both visualize specialization declarations and create a specializer for a given program. The other interface acts as a black box through which users can create specialized implementations.

Overall, the focus of this work is on the information needed to allow the program developer to declare what should occur during specialization, and on providing a convenient means for the program developer to supply this information. We also show how a partial evaluator that provides suitably precise analyses can use this information in its analysis phases and give appropriate feedback to the user regarding any inconsistencies between the inferred and declared binding times.

Our approach has been used in a course on partial evaluation for beginning graduate students, by engineers at Philips to specialize code related to digital television and by a systems programmer developing Forward Error Correction encoders (Le Meur et al., 2002). Despite having no previous experience in partial evaluation, our graduate students were able to quickly write specialization modules for classical partial evaluation examples, and to obtain the desired specialized code. They had a significantly easier time using specialization than students in previous years. The engineers also had no previous experience with partial evaluation, but were able to specialize the XDR library of RPC (Sun Microsystems, 1990) using our language after only a one-hour presentation of our language. The systems programmer had a similar experience in applying specialization to create many variants of a generic Forward Error Correction encoder.

The rest of this paper is organized as follows. First, we define the declaration language in Section 2. Section 3 illustrates how these declarations can help the program developer obtain the desired degree of specialization. Then, Section 4 describes the compilation of the specialization declarations and the graphical interfaces. Section 5 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 6. Finally, Section 7 concludes and suggests future work.

2. Declaration Language

Applying specialization to an already-developed program is a difficult task. The program developer first has to study the code to identify code fragments that contain interesting specialization opportunities, and then has 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, the program can be coded using strategies that are known to specialize well, and 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 developer to clearly describe the scenarios in which specialization is beneficial.

In this section, we first present the design decisions of our language. We then show the complete syntax and illustrate, through a simple example, the declarations related to the specification of binding-time constraints. Finally, we present the rest of the declarations, which describe the initialization context.

2.1. Design Decisions

In this section, we first give a brief overview of the partial evaluation process and then present our design decisions concerning the kinds of users to whom our specialization language is targeted, the kinds of declarations provided by our language, the granularity of these declarations, and the syntax used to provide these declarations.

Partial evaluation is often implemented as a two-stage process, consisting of an analysis phase followed by a specialization phase. The analysis phase identifies the computations that can be simplified based on knowledge of the provided inputs. The result of this analysis can be seen as a *specializable component*, that when given concrete values carries out the specialization phase. Guided by the results of the analysis phase, the specialization phase constructs an instance of the source program that is specialized to the provided values.

The analysis and specialization phases differ significantly in the amount of knowledge about the program that they require. The analysis phase completely determines the extent to which known values are propagated during specialization. Thus, successful use of the analysis phase requires identifying specialization opportunities and structuring the program such that these opportunities can be exploited by a partial evaluator. These operations require a deep understanding of the implementation, and thus we assume that the analysis phase is invoked

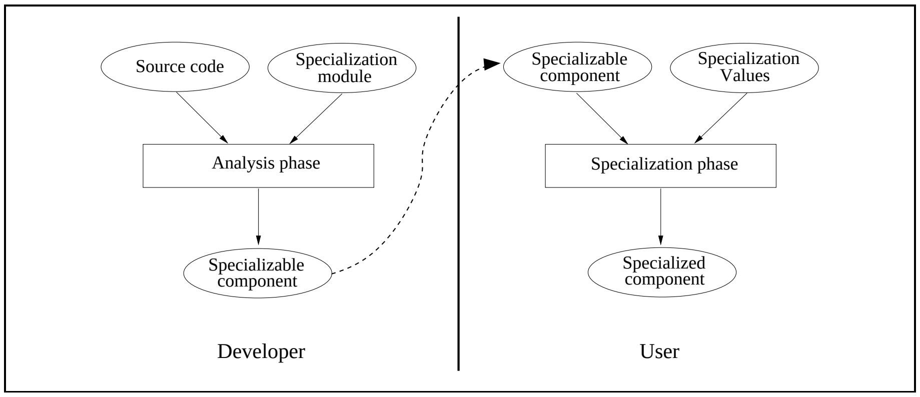


Figure 1. Specialization phases

by the program developer (henceforth, the *developer*). Choosing the specialization values, however, requires only a knowledge of how the specialized code will be used. Accordingly, we say that specialization is invoked by a *user*. Because of this difference in the levels of expertise, our approach is divided into two parts. The developer uses our declaration language to declare specialization scenarios that describe where known values should propagate throughout the program. The user, on the other hand, simply uses a GUI that requests the specialization values using descriptions provided by the developer, and requires no understanding of the program's implementation. The specialization process, including the roles of the developer and user, are illustrated in Figure 1.

As an example of the different roles of the developer and user, consider the traditional development and use of a highly optimized mathematical library. The developer chooses the algorithms, and uses as many implementation tricks as possible to create a highly optimized implementation. The user simply chooses a function from the library based on its functionality. The library may provide implementations that are optimized for particular inputs or ranges of inputs. Based on the documentation, the user may choose such an implementation rather than the generic one, but need to have no knowledge of the details of the optimization techniques that are used to create the chosen implementation.

At a minimum, the description of a specialization scenario must declare the point in the program (typically the name of a function) at which specialization should begin and the variables with respect to which the code should be specialized. Nevertheless, experience has shown that more information is required to ensure that an automatic partial evaluator can carry out the developer's intentions. Typically,

```
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();
  for(i = 0; i < u->length; i++)
    sum += u->data[i] * v->data[i];
  return sum;
}
```

Figure 2. The `dot` function

only a portion of a program can benefit from specialization. Thus, information is needed about which functions should be specialized and which should be considered as external. To ensure that specialization achieves the desired degree of optimization, information is needed about expected binding times within the code to be specialized. Finally, information is needed about how static values should be obtained.

As an example, consider the function `dot` shown in Figure 2. This function implements the multiplication of two integer vectors. It first checks that the vectors have the same size, aborting using the function `error` (not shown) if they do not, and then computes the dot product. Consider specialization of this function with respect to a completely static vector `u`. Such specialization essentially unrolls the `for` loop and replaces the references to the elements of `u` by constants. The minimum information needed to prepare this specialization is the function name `dot` and the information that `u`, on entry to this function, is completely static. The developer's intuition about how this specialization should be carried out, however, contains more information. For example, the developer might intend that the function `error` not be specialized, that the data values of the vector `u` remain static throughout the `dot` computation, and that these values be presented in a specific manner, *e.g.* entered manually by the user, or computed using some external function. The goals of our specialization scenario language are thus to allow the developer to express these properties and to do so in an intuitive manner.

There are several possible strategies to allow the developer to specify expected binding-times. One approach is for the developer to simply annotate every construct with its binding-time, thus reimplementing the source program in a *two-level language* (Engler et al., 1996; Taha and Sheard, 1997). This approach, however, seems burdensome for large programs. Instead, we propose that the developer declare the binding-time properties of *specialization parameters*: global variables, data-structure fields, and function parameters. The binding times of specialization parameters are then fixed according to these declarations at each reference point throughout the program, thus allowing pervasive checking of the developer's intentions. An important further practical advantage of this approach is that it implies that the declarations are not directly tied to the internal structure of each function definition. Most binding-time improvements described in the literature involve modifications to the structure of the source code, rather than to the set of specialization parameters (Jones et al., 1993; Jones, 1996; Thibault et al., 2000). Thus, most of the modifications required to "debug" the specialization process (*i.e.*, to achieve results from the analysis phase that will cause specialization to perform the desired propagation of the known information) do not require explicit adjustments to binding-time annotations.

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 specialization parameter in a single module. The result of preparing a program for specialization according to a selection of specialization scenarios forms a single specializable component.

2.2. Specialization context

We begin by considering the problems of specifying the code to be specialized and the binding times expected at intermediate points within this code. We first illustrate our approach with an example that highlights many features of the specification language, and then present the language in more detail.

```
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 3. Specialization module `vector`

2.2.1. *Example*

Consider again the function `dot` of Figure 2. This function is an example of a function that presents several specialization opportunities. If the sizes of both vectors are static, the size test can be reduced. If the size of the vector `u` is static, the `for` loop can be unrolled. Finally, 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 3, describes these specialization opportunities. The scenarios associated with the data structure `vec` and the function `dot` are:

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 field `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 field `data`, representing the data of the vector, is declared to have the type

`D(int *)`, indicating that the data should be considered 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 `dot` 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.

2.2.2. *Specification Language*

The complete syntax of the language of specialization declarations is defined in Figure 4. 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 with a specialization parameter defined in the source file indicated by the declaration `From` *file*. The basic unit of declaration is *bt_info_decl*, which has the form of a C declaration where the type is

$$
\begin{array}{lcl}
\mathit{module} & ::= & \mathtt{Module}\ \mathit{module\_id}\ \mathtt{\{}\,(\mathit{imports})^{?}\ \mathit{defines}\ \ \mathit{exports}\,\mathtt{\}} \\
\mathit{imports} & ::= & \mathtt{Imports}\ \mathtt{\{}\,(\mathtt{From}\ \mathit{file}\ \ \mathtt{\{}\,(\mathit{scen\_id}\,\mathtt{;})^{+}\,\mathtt{\}})^{*}\,\mathtt{\}} \\
\mathit{defines} & ::= & \mathtt{Defines}\ \mathtt{\{}\,(\mathtt{From}\ \mathit{file}\ \ \mathtt{\{}\,(\mathit{definition}\,\mathtt{;})^{+}\,\mathtt{\}})^{+}\,\mathtt{\}} \\
\mathit{exports} & ::= & \mathtt{Exports}\ \mathtt{\{}\,(\mathit{scen\_id}\,\mathtt{;})^{+}\,\mathtt{\}} \\
 & & \\
\mathit{definition} & ::= & \mathit{global\_def}\ \mid\ \mathit{struct\_def}\ \mid\ \mathit{proc\_def} \\
\mathit{global\_def} & ::= & \mathit{scen\_id}\,\mathtt{::}\,\mathit{bt\_info\_decl} \\
 & & \quad (\mathtt{\{}\,(\mathtt{needs}\ \mathtt{\{}\,(\mathit{scen\_id}\,\mathtt{;})^{+}\,\mathtt{\}})^{?}\ \ (\mathtt{inits}\ \mathtt{\{}\,(\mathit{init\_info\_decl}\,\mathtt{;})^{+}\,\mathtt{\}})^{?}\,\mathtt{\}})^{?} \\
\mathit{struct\_def} & ::= & \mathit{scen\_id}\,\mathtt{::struct}\ \ \mathit{struct\_id}\ \ \mathtt{\{}\,(\mathit{bt\_info\_decl}\,\mathtt{;})^{+}\,\mathtt{\}} \\
\mathit{proc\_def} & ::= & \mathit{scen\_id}\,\mathtt{::intern}\ \ \mathit{proc\_id}\,\mathtt{(}\,(\mathit{params})^{?}\,\mathtt{)} \\
 & & \quad (\mathtt{\{}\,(\mathtt{needs}\ \mathtt{\{}\,(\mathit{scen\_id}\,\mathtt{;})^{+}\,\mathtt{\}})^{?}(\mathtt{inits}\ \mathtt{\{}\,(\mathit{init\_info\_decl}\,\mathtt{;})^{+}\,\mathtt{\}})^{?}\,\mathtt{\}})^{?} \\
 & \mid & \mathit{scen\_id}\,\mathtt{::extern}\ \ (\mathit{bt\_info})^{?}\ \ \mathit{proc\_id}\,\mathtt{(}\,(\mathit{params})^{?}\,\mathtt{)} \\
\mathit{params} & ::= & (\mathit{bt\_info\_decl}\,\mathtt{,}\,)^{*}\ \ \mathit{bt\_info\_decl} \\
 & & \\
\mathit{bt\_info\_decl} & ::= & \mathit{base\_type}\ \ (\mathit{pointer})^{*}\ \ \mathit{id}\ \ (\mathit{array})^{*} \\
 & \mid & \mathtt{(}\,(\mathit{pointer})^{+}\ \ \mathit{id}\ \ (\mathit{array})^{*}\mathtt{)}\ \ \mathtt{()} \\
\mathit{base\_type} & ::= & \mathit{base\_bt}\,\mathtt{(}\mathit{simple\_type}\mathtt{)}\ \mid\ \mathit{struct\_type} \\
\mathit{base\_bt} & ::= & \mathtt{S}\ \mid\ \mathtt{D} \\
\mathit{simple\_type} & ::= & \mathtt{int}\ \mid\ \mathtt{long}\ \mid\ \mathtt{char}\ \mid\ \mathtt{...} \\
\mathit{pointer} & ::= & \mathit{base\_bt}\,\mathtt{(*)} \\
\mathit{array} & ::= & \mathit{base\_bt}\,\mathtt{([])} \\
\mathit{struct\_type} & ::= & \mathit{scen\_id}\,\mathtt{(struct}\ \ \mathit{struct\_id}\mathtt{)} \\
 & & \\
\mathit{init\_info\_decl} & ::= & \mathit{spe\_param}\ \ \mathtt{=}\ \ \mathtt{none} \\
 & \mid & \mathit{spe\_param}\ \ \mathtt{=}\ \ \mathit{expr} \\
 & \mid & \mathit{spe\_param}\ \ \mathtt{=}\ \ \mathit{proc\_call}\ \mathtt{From}\ \mathit{file} \\
\mathit{proc\_call} & ::= & \mathit{proc\_id}\,\mathtt{(}\,((\mathit{expr}\,\mathtt{,}\,)^{*}\mathit{expr})^{?}\,\mathtt{)} \\
 & & \\
\mathit{module\_id} & \in & \text{Module names} \\
\mathit{scen\_id} & \in & \text{Specialization scenario names} \\
\mathit{proc\_id} & \in & \text{Function names} \\
\mathit{struct\_id} & \in & \text{Structure names} \\
\mathit{id} & \in & \text{Identifier names} \\
\mathit{spe\_param} & \in & \text{Specialization parameters}
\end{array}
$$

Figure 4. Syntax of the specification language

decorated with either simple binding times (`S` or `D`) or, in the case of a structure type, with the name of a specialization scenario:

$$\begin{array}{lll}
\textit{bt\_info\_decl} & ::= & \textit{base\_type}\ \ (\textit{pointer})^*\ \ \textit{id}\ \ (\textit{array})^* \\
 & | & \texttt{(}\,(\textit{pointer})^+\ \ \textit{id}\ \ (\textit{array})^*\texttt{)}\ \ \texttt{()} \\
\textit{base\_type} & ::= & \textit{base\_bt}\texttt{(}\textit{simple\_type}\texttt{)}\ \ |\ \ \textit{struct\_type} \\
\textit{pointer} & ::= & \textit{base\_bt}\texttt{(*)} \\
\textit{array} & ::= & \textit{base\_bt}\texttt{([])} \\
\textit{base\_bt} & ::= & \texttt{S}\ \ |\ \ \texttt{D}
\end{array}$$

These declarations must be well-formed: a dynamic pointer cannot be declared to point to a static value. Scenarios can be declared for the following kinds of specialization parameters:

Global variables: The declaration

$$\textit{scen\_id}\,\texttt{::}\,\textit{bt\_info\_decl}$$

creates a specialization scenario *scen_id* that associates specialization information with the global variable mentioned in *bt_info_decl*.

Data structures: The declaration

$$\textit{scen\_id}\,\texttt{::struct}\ \textit{struct\_id}\ \texttt{\{}(\textit{bt\_info\_decl}\,\texttt{;})^+\texttt{\}}$$

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

Functions: The declaration

$$\textit{scen\_id}\,\texttt{::intern}\ \ \textit{proc\_id}\texttt{(}\,(\textit{params})^?\,\texttt{)}(\texttt{\{}\,\textit{needs\_list}\,\texttt{\}})^?$$

creates a specialization scenario *scen_id* that associates specialization information with each parameter of the function *proc_id*. The keyword `intern` indicates that the function should be specialized. In this case, the scenario must declare the scenarios of any functions, global variables, and structure types referred to by the procedure, using the *needs_list*.

The declaration

$$\textit{scen\_id}\,\texttt{::extern}\ \ (\textit{bt\_info})^?\ \ \textit{proc\_id}\texttt{(}\,(\textit{params})^?\,\texttt{)}$$

creates a specialization scenario *scen_id* for a function *proc_id* that should not be specialized. The scenario *scen_id* specifies that the parameters of the function *proc_id* should have the binding times described by *params*. Additionally, the optional declaration *bt_info* describes the binding time of the return value of *proc_id*. The binding times of the parameters and the return binding time control whether the external function is called during specialization. Specifically, the function is only called if all of the parameters are static and the return binding time, if provided, is static. For an external function, a *needs_list* is not needed, because the definition of the function is not seen by the partial evaluator.

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.

The syntax of the specialization module language implies that 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.

2.3. Initialization context

In addition to the binding-time constraints, the specialization module language permits the developer to specify the means of initializing static variables.

In most cases, the developer wants the specialization value of a static global variable or entry-point parameter to be set by the user at specialization time. In some cases, however, the value of a static parameter depends in a fixed way on the values of other static inputs. In these cases, the static parameter should not be exposed to the user. Instead, the initialization can be specified to occur either before entering the code to be specialized or within this code itself.

As an example, consider again the function `dot` shown in Figure 2. Suppose `dot` is used to compute the dot product of an input vector `v` of static size with another vector `u` of the same size, where `u` has some fixed structure. In this case, the means of computing `u` is a fixed part of the specialization context, and thus should be specified by the developer in the specialization scenario. The value of the size of `v`, on the other hand, is to be provided by the user at specialization time.

The default behavior is to ask the user for the initial values of static parameters. No declaration is needed in this case. To select between initialization prior to entering the code to be specialized and initialization within the code itself, the specification language provides three parameter-initialization scenario declarations:

$$\begin{array}{lll}
\mathit{init\_info\_decl} & ::= & \mathit{spe\_param}\ \ \texttt{=}\ \ \texttt{none} \\
 & \mid & \mathit{spe\_param}\ \ \texttt{=}\ \ \mathit{expr} \\
 & \mid & \mathit{spe\_param}\ \ \texttt{=}\ \ \mathit{proc\_call}\ \texttt{From}\ \mathit{file} \\
\mathit{proc\_call} & ::= & \mathit{proc\_id}\texttt{(}\,((\mathit{expr}\,\texttt{,}\,)^{*}\mathit{expr})^{?}\,\texttt{)}
\end{array}$$

The declaration *spe_param* `= none` means that the specialization parameter *spe_param* is initialized within the function body. The declaration *spe_param* `=` *expr* initializes *spe_param* with the value of *expr* at specialization time. The declaration *spe_param* `=` *proc_call* `From` *file* triggers *proc_call* to be executed at specialization time, before beginning specialization. This function should return the value to which the static parameter will be initialized. It is assumed that the initialization does not create aliases among the static inputs.

We use these declarations to express the desired specialization-time variable initialization context for `dot` as shown in the scenario `Btdot3` in Figure 5. The declaration line (1) specifies that the length of the vector `u` gets its value at specialization time from the length of the vector `v`. The field `data` of the vector `u` is set by calling the function `create_data`, which is defined in the file `init.c`, with `v->length` as an argument (line (2)). No initialization information is provided for the vector `v`, implying that its static field `length` will be set by the user.

3. Specialization Modules in Practice

In general, before specializing a program, the developer has an idea of the effect that should be produced by specialization. Ideally, all parts of the program that are not needed for the intended usage context should be eliminated, calculations that depend only on the static entry-point parameters should be simplified, and all occurrences of the static entry-point parameters should disappear. Overall, the specialized program should be more efficient.

Unfortunately, underspecialization and overspecialization occur frequently. In the case of underspecialization, static values are not propagated to the extent expected, and the resulting program is only partially specialized, or at worst simply identical to the source program. In the case of overspecialization, static values are propagated to contexts where they do not lead to further simplifications; the generation of many variants that are identical except for some constants leads to code explosion. In extreme cases, the specialization process does not terminate.

```
Module vector {
 ...
 Defines {
  From dotproduct.c {
   VecDS::struct vec
          {D(int *) data; S(int) length;};
   VecSS::struct vec
          {S(int *) data; S(int) length;};
   ...
   Bterr::extern error();
   ...
   Btdot3::intern dot
           (VecSS(struct vec) S(*) u,
            VecDS(struct vec) S(*) v)
           { needs{Bterr;}
             inits{ u->length = v->length;           (1)
                    u->data = create_data(v->length)
                              From init.c; }         (2)
 ...}...}
 Exports {VecDS; VecSS; ...; Btdot3;...}
}
```

Figure 5. Specialization module `vector` with initialization declarations

We examine some typical cases of underspecialization and overspecialization, and show how the use of specialization modules addresses these issues.

3.1. Underspecialization

Underspecialization occurs when locations (variables or memory locations) intended to be static are considered to be dynamic by the binding-time analysis. Once dynamic, the values of these locations cannot be used during specialization, and less simplification occurs than expected.

There are several ways in which a location can become dynamic. The most obvious is simply the assignment of the location to the value of a dynamic expression. More difficult to detect are the approximations due to dynamic control and data flow. For example, all locations assigned in the body of a loop having a dynamic test or in the branches of a conditional having a dynamic test are considered dynamic subsequent to the loop or conditional. When a conditional has a static test, the binding time of each location assigned in the conditional is a safe approximation of the binding times of the location at the end of each branch. Thus, a location becomes dynamic after such a conditional if it is dynamic in only one of the branches. Finally, locations become dynamic because of limits in the granularity of the binding-time analysis. For example, in Tempo, when one array element is assigned to a dynamic value, all other elements of the same array are subsequently considered dynamic.

To illustrate the use of specialization modules in detecting the reasons for underspecialization, we consider specialization of the Berkeley Packet Filter (BPF) interpreter, `bpf_filter` (McCanne and Jacobson, 1993), of which an excerpt is shown in Figure 6. The arguments to the interpreter are a pointer `pc` to the current instruction in the program, a packet `p`, the total size `wirelen` of the packet, and the size `buflen` of the portion of the packet that has not yet been treated. The interpreter is implemented as a `while` loop, of which each iteration decodes the current instruction, executes it, possibly stores the result in the temporary variable `A`, and increments `pc`. This process is repeated for each packet. Because the program is fixed across the treatment of a sequence of packets, specialization of the interpreter with respect to the program should improve performance. Indeed, specialization of an interpreter with respect to a program is a classic example of the benefits of partial evaluation, and specialization of the BPF interpreter has previously been considered (McNamee et al., 2001).

Unfortunately, using Tempo to specialize the BPF interpreter shown in Figure 6 with respect to the value of `pc` produces no benefit whatsoever. Independently of the use of specialization modules, the binding-time analysis of Tempo provides some feedback to the developer, in the form of a copy of the source file in which each construct is colored according to its binding time. Examination of the colored file produced for the BPF interpreter indicates that `pc` is considered dynamic throughout the interpretation loop. Because a loop is analyzed by a fixpoint iteration that propagates the binding time of each location at the end of the loop back to the beginning of the loop, the knowledge that `pc` is dynamic throughout the loop is, however, not sufficient to identify the point at which it first receives a dynamic binding time.

To detect the source of the problem, we construct a specialization module declaring the intended specialization context. The specialization module `bpf`, shown in Figure 7, contains two scenarios: `Btstruct`, which declares an instance of the `bpf_insn` structure (the type of `pc`) in which all of the fields are static, and `Btbpf`, which declares that the `pc` argument should be a static pointer to such a completely static

```
struct bpf_insn {
  u_short code;
  u_char jt;
  u_char jf;
  int k;
};

int bpf_filter(struct bpf_insn *pc, u_char *p,
               u_int wirelen, u_int buflen) {
  int A;
  ...
  while(1) {
    switch(pc->code) {
      case ...
      case BPF_JMP|BPF_JGT|BPF_K:
        if (A > pc->k)
          pc = pc + pc->jt;
        else
          pc = pc + pc->jf;
        break;
      case ...
    }
    pc = pc + 1;
  }
}
```

Figure 6. Excerpt of the BPF interpreter

`bpf_insn` structure and that the other arguments, which represent information about the packet to be processed, should be dynamic.

Given the specialization module `bpf`, analysis of the `bpf_filter` function gives an error at the point of the assignment `pc = pc + 1`. The error message indicates that the value of `pc` in the increment expression `pc + 1` is dynamic, contradicting the declaration of `pc` as static in the scenario `Btbpf`. The fact that the binding-time mismatch is first detected at a reference to `pc` rather than at an assignment to `pc` indicates that the dynamic binding time is due to the effect of dynamic control or data flow, rather than an explicit assignment to `pc`. Because the precision of the binding-time analysis of Tempo is sufficient that `pc` is given an individual binding time, and because no other variable is an alias of `pc`, we focus on control flow operations.

```
Module bpf {
  Defines {
    From bpf_filter.c {
      Btstruct::struct bpf_insn {
                      S(u_short) code;
                      S(u_char)  jt;
                      S(u_char)  jf;
                      S(int)     k;
                    };
      Btbpf::intern bpf_filter
                (Btstruct(struct bpf_insn *) pc,
                 D(u_char *)                 p,
                 D(u_int)                    wirelen,
                 D(u_int)                    buflen);
    }
  }
  Exports{Btbpf;}
}
```

Figure 7. Specialization module for the BPF interpreter

To explore the problem further, we place references to `pc`, amounting to binding-time guards, following each dynamic conditional. Repeating the binding-time analysis then indicates the points at which the treatment of variables assigned within the branches of a dynamic conditional causes `pc` to become dynamic. This approach indicates that the first point at which an inferred binding time is not compatible with the declared binding times is just after the conditional:

```
if (A > pc->k) pc = pc + pc->jt; else pc = pc + pc->jf;
```

in the treatment of a `BPF_JMP|BPF_JGT|BPF_K` instruction. The test depends on `A`, whose value is determined by applying some instruction to the dynamic packet, and is thus dynamic. Although the assignments to `pc` are initially static within the branches of this conditional, the dynamic test causes `pc` to be considered dynamic thereafter.

Having detected the source of the problem, the developer can try to reorganize the program such that `pc` remains static. Such binding-time improvement remains an art, but useful strategies are documented in the literature (Jones et al., 1993; Jones, 1996; Thibault et al., 2000). When a location undesirably becomes dynamic due to an assignment in a dynamic conditional, a standard binding-time improvement is to propagate the complete context of the dynamic conditional (*i.e.*,

```
int bpf_filter(struct bpf_insn *pc, u_char *p,
               u_int wirelen, u_int buflen) {
  int A;
  ...
  while(1) {
    switch(pc->code) {
      case ...
      case BPF_JMP|BPF_JGT|BPF_K:
        if (A >= pc->k)
          return bpf_filter(pc + pc->jt + 1, p,
                            wirelen, buflen);
        else
          return bpf_filter(pc + pc->jf + 1, p,
                            wirelen, buflen);
      case ...
    }
    pc = pc + 1;
  }
}
```

Figure 8. Binding-time improvement of the BPF interpreter

the *continuation* (Plotkin, 1975)) into both branches of the conditional (Consel and Danvy, 1991). In the case of `bpf_filter`, we can achieve this propagation by implementing each branch as a recursive call, which has the effect of performing all of the remaining iterations of the loop (McNamee et al., 2001). The modified program is shown in Figure 8. With this modification of the program, the binding-time analysis of Tempo determines that `pc` is static throughout the program. Specialization of the modified BPF interpreter with respect to a program produces a direct implementation of a filter, containing no interpretive overhead.

As illustrated by this example, the use of specialization modules helps identify the source of a binding-time problem, but does not address the issue of how to reorganize the program to solve the problem. In our experience, users can become familiar with binding-time improvements relatively quickly; finding the source of a binding-time problem is the most difficult. This observation is particularly true in the case of a program that uses loops or recursion heavily, because of the effect of the fixpoint iteration in the analysis. In this case, the use of specialization modules can simplify the use of partial evaluation for both novice users and experts.

In the approach presented here, the developer inserts dummy expressions into the program to localize binding-time problems. Automatically adding such expressions for all locations at all conditionals could substantially increase the binding-time analysis time and signal binding-time conflicts that do not actually show up in the original program, *e.g.* when a subsequent assignment restores the correct binding time of the location before the next reference in the original program. Nevertheless, the specialization module compiler could straightforwardly be extended to insert such expressions for a developer-specified location after each dynamic control-flow construct in the scope of the location. Alternatively, the analysis could be extended to record with each dynamic location a pointer to the expression that most recently caused the binding time of the location to change from static to dynamic. Ultimately, a debugging environment analogous to `gdb` could be constructed to allow the developer to dynamically insert and remove such guards, and thus monitor the binding-time analysis process.

3.2. Overspecialization

Overspecialization occurs when a function is specialized with respect to static values that do not induce useful specialization-time computations. To illustrate overspecialization, we use the function `find`, considered by Neil Jones in the paper “What *Not* to Do When Writing an Interpreter for Specialisation” (Jones, 1996). As shown by Jones, specialization causes this function to increase in size excessively when specialized with the C-Mix partial evaluator (Andersen, 1994; C-Mix, 2000).

The function `find`, shown in Figure 9, implements an iterative binary search for an element `x` in an array `tab` of size twice `delta`. The remaining argument `i` records the current position in the array on each iteration. A typical use of `find` is illustrated by the function `binsearch`, also shown in Figure 9, in which the argument `i` is initialized to 0.

Using C-Mix, specialization of `binsearch` with respect to `delta = 4` produces the program shown in Figure 10. The size of the specialized program is linear in the size of the array `tab`, which, as observed by Jones, is unacceptable for large arrays. Indeed, much of the code growth is due to the simple instantiation of array references and `return` statements with various values of `i`.

The source of the code growth lies in the specialization strategy of C-Mix, which propagates the context of each dynamic conditional into each branch, thus automatically achieving the effect we obtained by

```
int find(int *tab, int i, int delta, int x) {
  loop:
  if (delta == 0) {
    if (x == tab[i])
      return(i);
    else return(NOTFOUND);
  }
  if (x >= tab[i+delta]) i = i + delta;
  delta = delta/2;
  goto loop;
}

int binsearch(int *t, int delta, int x) {
  return find(t, 0, delta, x);
}
```

Figure 9. Definition of `find` and `binsearch`

manual transformation of the BPF interpreter in Section 3.1. In the case of `find`, however, the propagation is not desirable, as it uselessly propagates each possible value of `i` to succeeding loop iterations. The solution is simply to use a specialization module to declare `i` to be dynamic, as shown in Figure 11. When `i` is considered as dynamic, the context of each conditional specializes identically with respect to the state produces by each branch, and the memoization strategy of C-Mix prevents code duplication. The result of specialization when `delta` is 4 is shown in Figure 12. In general, the size of the specialized program is proportional to the log of the size of the processed array. Because the aggressive propagation strategy of C-Mix risks frequently producing this kind of overspecialization, C-Mix provides features for specifying the binding-times of individual variables, but without the modularity provided by our approach.

Overspecialization can also occur when a function is called within a static loop. In this case, specialization typically generates a specialized instance of the function for each loop iteration, which is undesirable if no interesting computation depends on the loop index. Using a specialization module to indicate that some parameters of the called function should be considered dynamic is typically sufficient to solve the problem.

In some cases, specialization with respect to a static value is desirable when the value is small, but causes excessive code growth when the

```
int find_spe(int *tab, in x) {
  if (x >= tab[4])
    if (x >= tab[6])
      if (x >= tab[7]) {
        if (x == tab[7])
          return 7;
      }
      else {
        if (x == tab[6])
          return 6;
      }
    else
      if (x >= tab[5]) {
        if(x == tab[5])
          return 5;
      }
      else {
        if (x == tab[4])
          return 4;
      }
  else
    if (x >= tab[2])
      if (x >= tab[3]) {
        if (x == tab[3])
          return 3;
      }
      else {
        if (x == tab[2])
          return 2;
      }
    else
      if (x >= tab[1]) {
        if (x == tab[1])
          return 1;
      }
      else {
        if (x == tab[0])
          return 0;
      }
  return NOTFOUND;
}

int binsearch_spe(int *t, int x) {
  return find_spe(t,x);
}
```

Figure 10. Specialization of `binsearch` with respect to `delta = 4`

```
Module search {
  Defines {
    From search.c {
      Btfind::intern find(D(int *) tab,
                          D(int)   i,
                          S(int)   delta,
                          D(int)   x);
      Btsearch::intern binsearch(D(int *) t,
                                 S(int)   delta,
                                 D(int)   x)
                  { needs { Btfind; } };
    }
  }
  Exports { Btsearch; }
}
```

Figure 11. Specialization module for `binsearch`

```
int find_spe(int *tab, in x) {
  if (x >= tab[i + 4])
    i = i + 4;
  if (x >= tab[i + 2])
    i = i + 2;
  if (x >= tab[i + 1])
    i = i + 1;
  if (x == tab[i])
    return i;
  else
    return NOTFOUND;
}
```

Figure 12. Specialization of `find` for `delta = 4` according to the specialization module `search`

value is large. To address this issue, the specialization module language allows a range test to be associated with a declaration that a scalar-typed location should be static. The specialization module compiler inserts a corresponding static test of the value into the source code. This test triggers a specialization-time error if the static value is out of the required range.

3.3. Assessment

We have seen that the use of specialization modules can help pinpoint the cause of underspecialization and can allow the developer to constrain the specialization process to avoid overspecialization. The examples we have considered illustrate problems that occur in specialization of a single module. In general, specialization modules for a complex application should be constructed incrementally, such that new specialization modules refer to existing specialization modules that are known to describe valid specialization scenarios. This approach further helps detect the source of unexpected binding times, by localizing the problem to the code controlled by the new specialization module. For example, when an interpreter is built on top of a virtual machine that itself presents specialization opportunities (Thibault et al., 1998), the specialization modules for the virtual machine can be developed and tested before those for the interpreter. Binding-time problems in specialization of the interpreter are thus confined to the definition of the interpreter itself and its interface with the virtual machine, but cannot derive from the implementation of the virtual machine.

4. Specialization Module Compilation

The declarations of a specialization module are connected to the specialization process by the specialization module compiler. The front end of this compiler analyzes the dependencies between the specialization scenarios, checks that scenarios are used correctly across the specialization modules (*e.g.,* only scenarios defined in the current module can be exported, only exported modules can be imported), and checks that the specialization modules are consistent with 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 back end of the compiler translates the specialization scenarios into configuration declarations specific to the targeted partial evaluator.

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. The compiler extracts the code to be specialized from the various files of the source program and reassembles it into a single C source file. The compiler also generates directives that inform Tempo of the binding-time properties of input values of complex type and of the results of external function calls, in accordance with the specialization module declarations. Furthermore, we have developed a graphical environment both to assist the program developer in the creation of specializable components, and to enable users to obtain specialized implementations by specifying specialization values.

Developer Interface The Developer Interface (Figure 13) allows the developer to prepare the specialization process. The interface initially presents a list of the specialization modules defined in the current directory. Selecting one of the modules displays a list of the scenarios it defines. Selecting one of these scenarios amounts to choosing this scenario as the entry-point of specialization. The commands provided by the interface allow the developer to visualize and modify the modules and scenarios, and to perform the preprocessing required to generate a specializable component.

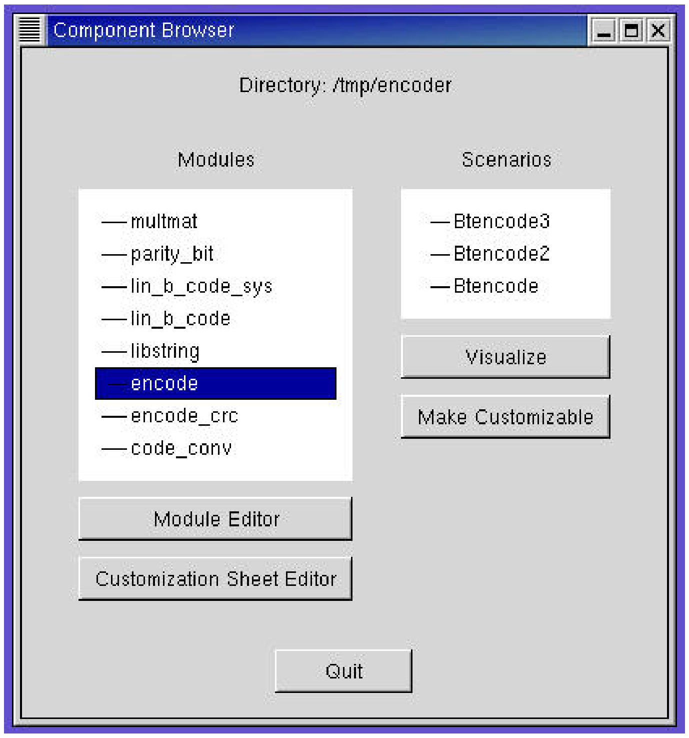
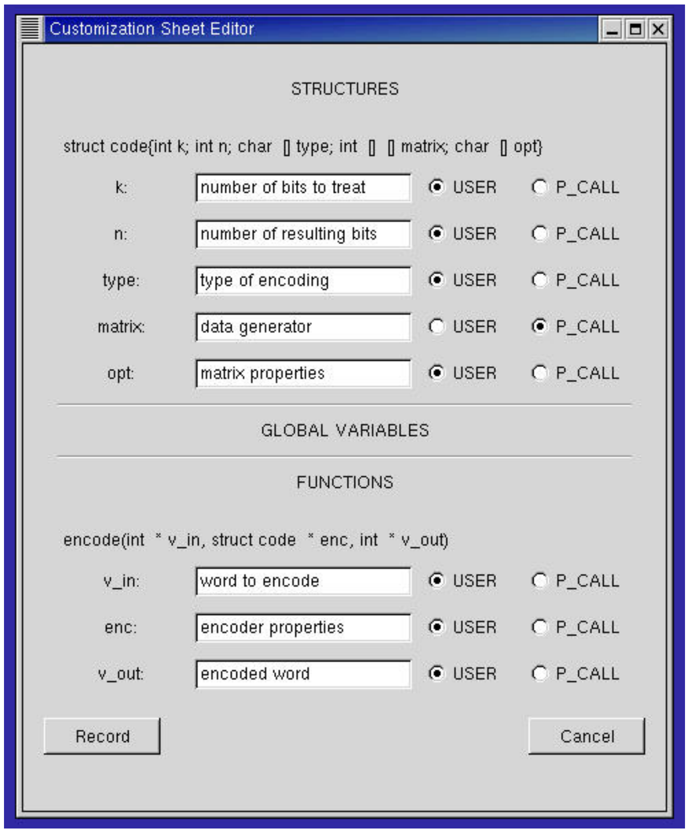


Figure 13. Developer interface

The Module Editor command and the Customization Sheet Editor command of the Developer Interface allow the developer to describe properties of the selected module as a whole. The Module Editor command launches a text editor containing the source code of the module; new scenarios can be added and existing scenarios can be modified or


Customization Sheet Editor

STRUCTURES

struct code{int k; int n; char [] type; int [] [] matrix; char [] opt}

k:	number of bits to treat	[✓ USER]	P_CALL
n:	number of resulting bits	[✓ USER]	P_CALL
type:	type of encoding	[✓ USER]	P_CALL
matrix:	data generator	USER	[✓ P_CALL]
opt:	matrix properties	[✓ USER]	P_CALL

GLOBAL VARIABLES

FUNCTIONS

encode(int * v_in, struct code * enc, int * v_out)

v_in:	word to encode	[✓ USER]	P_CALL
enc:	encoder properties	[✓ USER]	P_CALL
v_out:	encoded word	[✓ USER]	P_CALL

Figure 14. Customization sheet editor

deleted. Once editing is completed, the developer can save the module for later use. The Customization Sheet Editor command allows the developer to provide textual descriptions of all the structure fields, global variables, and function parameters that are referred to in the specialization module (Figure 14). These descriptions are then saved in a file to be used later when prompting the user of the specializable component for the actual input values.

The Visualize command of the Developer Interface generates the Component Dependencies window and the Scenario Dependencies window (both shown in Figure 15), which illustrate the relationship between the selected scenario and other scenarios. The Component Dependencies window presents a graph in which the nodes represent specialization modules and the edges summarize the “needs” relations between the scenarios defined by these modules. The Scenario Dependencies window presents dependency information at a finer grain of detail. Beginning with the selected scenario, the window presents a tree

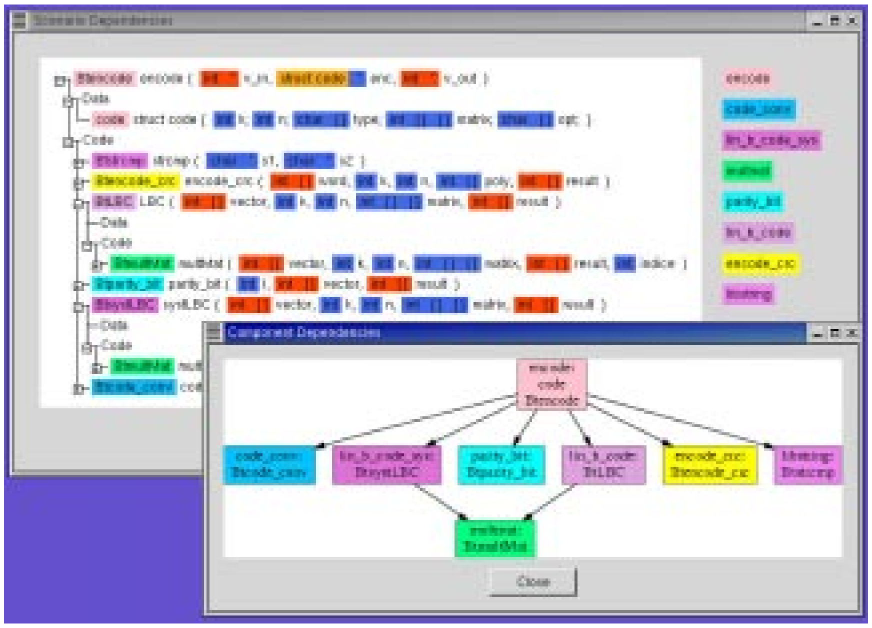


Figure 15. Visualization tools

of the referenced scenario definitions, which are classified according to whether they refer to data structures ("Data") or function definitions ("Code"). Within each scenario, types are color-coded to indicate the associated binding-time specifications.

Finally, the Make Customizable command of the Developer Interface collects the program fragments required for specialization, beginning with the specified scenario and following the "needs" declaration. The analysis phase of the partial evaluator is then applied to the resulting program. The result is a specializable instance of the operation indicated by the selected specialization scenario. The collection of instances created from the scenarios of a given specialization module as well as the textual description of all the global variables, data structures and function parameters involved, makes up a specializable component.

User Interface The User Interface (Figure 16) allows the component user to create specialized implementations of the operations provided by a specializable component. The interface presents the user with a list of specializable components. Selecting a component creates a list of the scenarios that have been prepared for specialization. Selecting a scenario generates the associated Component Customization Interface (shown in Figure 17), which prompts the user for the values of the static inputs, using the descriptions provided by the developer in the

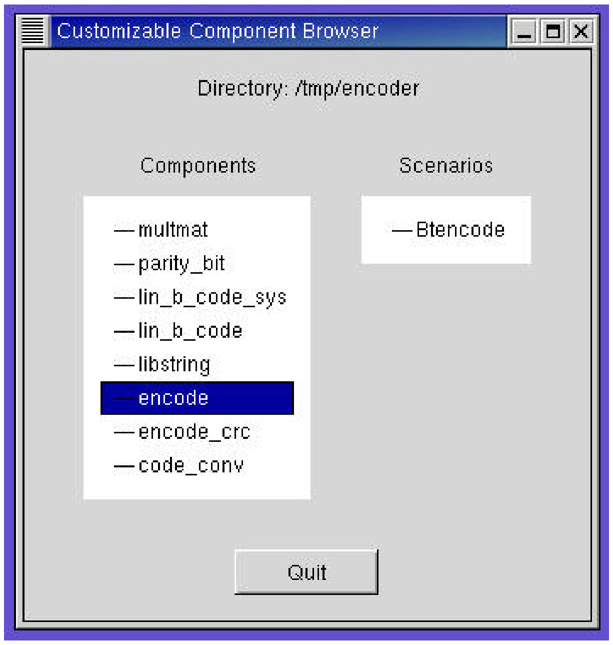
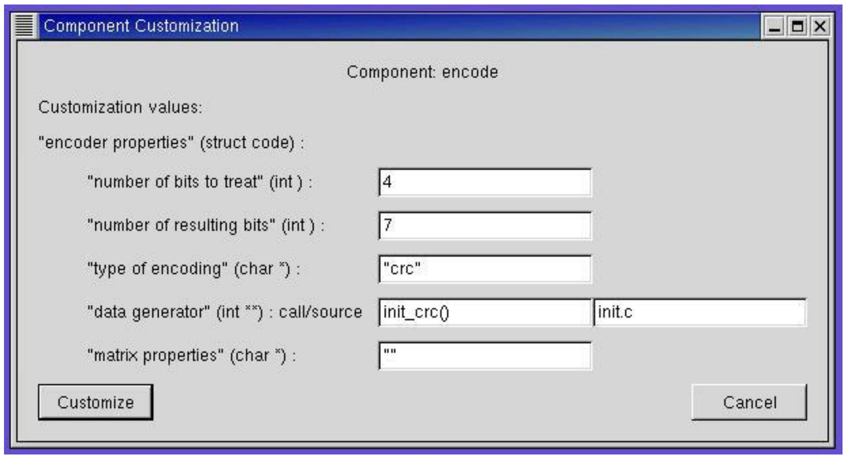


Figure 16. User interface



Component: encode

Customization values:

"encoder properties" (struct code) :

"number of bits to treat" (int) :	4	
"number of resulting bits" (int) :	7	
"type of encoding" (char *) :	"crc"	
"data generator" (int **) : call/source	init_crc()	init.c
"matrix properties" (char *) :	""	

Figure 17. Component customization interface

Component Sheet Editor. The Customize command of the Component Customization Interface then carries out the specialization.

5. 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.

Declarations are checked during the analysis phase of partial evaluation. We begin by describing a possible strategy to use for these checks. This strategy is appropriate for Tempo, but in general has to be adapted to the precision of the binding-time analysis of the target partial evaluator. We then consider specialization based on the results of such an analysis. In an offline partial evaluator, specialization simply follows the directions of the annotations produced by the binding-time analysis. We show that the specialization based on our modified binding-time analysis both respects the semantics of the source program (the standard correctness criterion for a partial evaluator) and respects the declarations, thus providing the desired degree of specialization.

5.1. Verification Strategy

The degree of specialization is determined by the binding times of references to locations (*i.e.*, variables and memory locations), because these are the points through which static values propagate within the program. Thus, the analysis of each variable reference and each dereference expression checks that the binding time matches the declarations on the possibly referenced locations. In other cases, the proposed verification strategy is more lazy, allowing the inferred binding time for a location to conflict with the specified binding time as long as the location is not explicitly referenced. This laziness increases the set of programs that are accepted for specialization, as compared to complete checking of all implicit effects, and ensures that error messages about incompatible binding times only refer to explicit program constructs.

For conciseness, the examples in this section 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). Static terms are underlined. The examples are quite simple in order to focus on specific aspects of the analysis.

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. The annotations on the function `f`, defined below, specify that both arguments should be static pointers, and that a dereference of `x` should be dynamic while a dereference of `y` should be static:

```
void f(int$^{\mathcal{D}}$ *$^{\mathcal{S}}$ x, int$^{\mathcal{S}}$ *$^{\mathcal{S}}$ y) {
  ...
}
```

Suppose `f` is called from the following context:

```
void main() {
  int$^{\mathcal{U}}$ $\underline{\texttt{a = 3;}}$
  f ($\underline{\texttt{\&a}}$, $\underline{\texttt{\&a}}$);
}
```

Here, both arguments to `f` 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 `int`$^{\mathcal{D}}$ `*`$^{\mathcal{S}}$ and the declaration of `y` as `int`$^{\mathcal{S}}$ `*`$^{\mathcal{S}}$.

Assignments: To verify that information is flowing through the program according to the developer's intentions, the analysis checks that the binding time of the right-hand side expression is compatible with the constraints on the possibly modified locations. Nevertheless, following a lazy strategy, the analysis does not check the compatibility of constraints and binding times at all levels of indirection for aliases implicitly affected by the assignment.

Verification of the assignment `y = &d` in the following example illustrates this laziness in the treatment of assignments.

```
int$^{\mathcal{D}}$ d;

void f(int$^{\mathcal{S}}$ *$^{\mathcal{S}}$ y) {
  $\underline{\texttt{y = \&d;}}$
}
```

The static binding time of `&d` is sufficient to satisfy the declaration of `y` as `int`$^{\mathcal{S}}$ `*`$^{\mathcal{S}}$. The laziness of the verification of assignments

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

implies that the mismatch between the dynamic binding time of `d` and the static constraint on `*y` is not taken into account, because the binding time of `*y` has no impact on specialization of the assignment statement.

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. The analysis is lazy in that it does not check this inferred binding time until such a location is actually referenced.

In the following example, `x` is assigned to `NULL` within a dynamic conditional statement.

```
int$^{\mathcal{D}}$ d;

void f(int$^{\mathcal{D}}$ *$^{\mathcal{S}}$ x, int$^{\mathcal{S}}$ *$^{\mathcal{S}}$ y) {
  if (d) {
    x = NULL;
    ...           /* may contain static uses of x */
  }
  x = y;
}
```

`NULL` is static, in accordance with the constraint on `x`, but following the conditional statement, `x` is considered dynamic. This adjustment of the binding time is not checked. The subsequent explicit assignment of `x` to `y` restores the correct binding time of `x`.

Variable reference or dereference expression: For a variable reference or a dereference expression, the analysis checks that the constraint on the location is compatible with its inferred binding time, but does not check such compatibility for all possible dereferences of the location. This laziness is illustrated by the reference to `y` in the following example:

```
int$^{\mathcal{D}}$ d;

void f(int$^{\mathcal{D}}$ *$^{\mathcal{S}}$ x, int$^{\mathcal{S}}$ *$^{\mathcal{S}}$ y) {
  y = &d;
  x = y;
}
```

Although the assignment to `y`, `y = &d`, causes the binding time of `*y` to conflict with the declaration on `y`, this declaration is not checked in the assignment `x = y`, 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. In the following example, at the point of the `return` statement, `*z` is an alias for `y` and `**z` is an alias for `d`, as indicated by the superscripted sets on these expressions below.[3] The dereference `**z` 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.

$$\begin{array}{l}
\mathtt{int}^{\mathcal{D}}\ \mathtt{d;} \\
\\
\mathtt{int\ f(int}^{\mathcal{S}}\ \mathtt{*}^{\mathcal{S}}\ \mathtt{y)\ \{} \\
\quad \mathtt{int}^{\mathcal{U}}\ \mathtt{*}^{\mathcal{U}}\ \mathtt{*}^{\mathcal{U}}\ \underline{\mathtt{z\ =\ \&y;}} \\
\quad \underline{\mathtt{y\ =\ \&d;}} \\
\quad \mathtt{return\ *(*z}^{\{\mathtt{y}\}}\mathtt{)}^{\{\mathtt{d}\}}\mathtt{;} \\
\mathtt{\}}
\end{array}$$

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 (Ashley and Consel, 1994), as done in our implementation.

5.2. Specialization

The specialization phase simplifies the static constructs and reconstructs the dynamic constructs to form the specialized program. Because binding-time declarations are completely encoded into the results of the analyses, the specialization modules 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

[3] 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.

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 Developer's Declarations: Intuitively, specialization that respects the developer'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. The proposed verification strategy thus allows 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. Static and dynamic contexts are defined more formally in the appendix.

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 evaluated expression can be considered to be filling the hole of a static or dynamic context respectively, ρ is a store mapping locations to values, and $\mathtt{x}_\ell$ is the location associated with the variable $\mathtt{x}$ (this location is determined implicitly). The complete instrumented semantics for the statements and expressions of our example imperative language is defined in the appendix.

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

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 during execution of the specialized program according to the annotated semantics, a location declared to be static is never referenced in a static context. Furthermore, if such a location has a liftable type (non-pointer type, for C programs), it is never referenced in a dynamic context. The proof follows from the fact that the treatment of non-liftable values in Tempo only reannotates a static expression having a non-liftable value when this expression occurs in a dynamic context.

6. Related Work

The difficulty of obtaining a desired specialized program by automatic techniques alone has led to a variety of approaches that allow the developer 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 (Volanschi et al., 1997), none of the previous strategies provide any structuring of the specialization declarations.

DyC The DyC run-time specialization system includes annotations that allow the developer to control various aspects of the specialization process, including the propagation of specialized values (Grant et al., 2000b). Based on developer annotations identifying static variables, DyC automatically infers the region of code to be specialized. Although the annotation language allows the developer 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 developer's annotations and the source code (Grant et al., 2000b).

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 (C-Mix, 2000). 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 apply to each kind object that can be referenced via a variable, as indicated by the variable's type, and are thus more fine-grained. 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 developer to specify which static arguments should be propagated into the body of a specialized function (Consel, 1993). Filters 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 developer 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 developer 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 (Lee and Leone, 1996). 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 Meta-ML (Taha and Sheard, 1997) and 'C (Engler et al., 1996), provide a high-level notation for describing the construction of code fragments. While such languages allow the construction of specialized code satisfying precise 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 form a declarative approach for specifying specialization opportunities in Java programs (Volanschi et al., 1997). Declarations focus on the integration of the specialized code into the original program. A specialization class indicates the names of the methods at which to begin specialization, identifies the arguments known at specialization time, 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, the developer cannot guide the propagation of static arguments and thus cannot control how the specialization process is carried out.

Our approach is complementary to the use of specialization classes in that our goal is to provide developer control over the specialization process itself. Like a specialization class, a specialization module allows the developer to describe the specialization context of the entry point. The declarations of the binding times of function parameters, global variables, and data structures in a specialization module allow the developer 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 (Constable et al., 1986), and is beginning to be recognized in the area of program analysis as well (Grobauer, 2001; Ryder, 1997).

7. Conclusion

We have presented a high-level language that enables the developer 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 main 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 predictability of the specialization process. Feedback provided when a declaration mismatch occurs helps identify specialization problems. 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.

We have successfully applied our approach to numerous existing real-sized partial evaluation examples such as the FFT (Lawall, 1998), the XDR library of RPC (Muller et al., 1997; Muller et al., 1998), the Berkeley packet filter (Thibault et al., 2000) and Unix signals (McNamee et al., 2001). Furthermore we have tested our language in both graduate teaching and industrial contexts. Overall, 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.

Our language permits the developer 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 begun to extend our declaration language to allow the developer 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.

References

Andersen, L.: 1994, 'Program Analysis and Specialization for the C Programming Language'. Ph.D. thesis, Computer Science Department, University of Copenhagen. DIKU Technical Report 94/19.

Andersen, P.: 1996, 'Partial evaluation applied to ray tracing'. In: W. Mackens and S. Rump (eds.): *Software Engineering in Scientific Computing*. pp. 78–85, Vieweg. DIKU Technical Report D-289.

Ashley, J. M. and C. Consel: 1994, 'Fixpoint Computation for Polyvariant Static Analyses of Higher-Order Applicative Programs'. *ACM Transactions on Programming Languages and Systems* **16**(5), 1431–1448.

Berlin, A. and R. Surati: 1994, 'Partial Evaluation for Scientific Computing: The Supercomputer Toolkit Experience'. In: *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. Orlando, FL, USA, pp. 133–141, Technical Report 94/9, University of Melbourne, Australia.

Blazy, S.: 2000, 'Specifying and Automatically Generating a Specialization Tool for Fortran 90'. *Journal on Automated Software Engineering* **7**(4), 345–376.

Bondorf, A.: 1992, 'Improving Binding Times Without Explicit CPS-Conversion'. In: *ACM Conference on Lisp and Functional Programming*. San Francisco, CA, USA, pp. 1–10, ACM Press.

C-Mix: 2000, 'C-Mix$_{/II}$ User and Reference Manual'. http://www.diku.dk/research-groups/topps/activities/cmix/download/.

Christensen, N. H., R. Glück, and S. Laursen: 2000, 'Binding-Time Analysis in Partial Evaluation: One Size Does Not Fit All'. *Lecture Notes in Computer Science* **1755**, 80–92.

Consel, C.: 1993, 'A Tour of Schism: a partial evaluation system for higher-order applicative languages'. In: *Partial Evaluation and Semantics-Based Program Manipulation*. Copenhagen, Denmark, pp. 66–77, ACM Press.

Consel, C. and O. Danvy: 1991, 'For a Better Support of Static Data Flow'. In: J. Hughes (ed.): *Functional Programming Languages and Computer Architecture*, Vol. 523 of *Lecture Notes in Computer Science*. Cambridge, MA, USA, pp. 496–519, Springer-Verlag.

Consel, C., L. Hornof, F. Noël, J. Noyé, and E. Volanschi: 1996, 'A Uniform Approach for Compile-Time and Run-Time Specialization'. In (Danvy et al., 1996), pp. 54–72.

Consel, C. and S. Khoo: 1993, 'Parameterized Partial Evaluation'. *ACM Transactions on Programming Languages and Systems* **15**(3), 463–493.

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

Danvy, O.: 1998, 'Type-Directed Partial Evaluation'. In (Hatcliff et al., 1998), pp. 367–411, Springer-Verlag.

Danvy, O., R. Glück, and P. Thiemann (eds.): 1996, 'Partial Evaluation, International Seminar, Dagstuhl Castle', No. 1110 in Lecture Notes in Computer Science.

Engler, D., W. Hsieh, and M. Kaashoek: 1996, ''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*. St. Petersburg Beach, FL, USA, pp. 131–144, ACM Press.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides: 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

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

Grant, B., M. Mock, M. Philipose, C. Chambers, and S. Eggers: 2000b, 'DyC: An Expressive Annotation-Directed Dynamic Compiler for C'. *Theoretical Computer Science* **248**(1–2), 147–199.

Grobauer, B.: 2001, 'Cost Recurrences for DML Programs'. In: *ICFP 2001: International Conference on Functional Programming*. Florence, Italy, pp. 253–264, ACM Press.

Hatcliff, J., T. Æ. Mogensen, and P. Thiemann (eds.): 1998, 'Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School', No. 1706 in Lecture Notes in Computer Science. Copenhagen, Denmark: Springer-Verlag.

Hornof, L. and J. Noyé: 2000, 'Accurate Binding-Time Analysis for Imperative Languages: Flow, Context, and Return Sensitivity'. *Theoretical Computer Science* **248**(1–2), 3–27.

Hughes, J.: 1998, 'A Type Specialisation Tutorial'. In (Hatcliff et al., 1998), pp. 293–325, Springer-Verlag.

Jones, N.: 1996, 'What *Not* to Do When Writing an Interpreter for Specialisation'. In (Danvy et al., 1996), pp. 216–237.

Jones, N., C. Gomard, and P. Sestoft: 1993, *Partial Evaluation and Automatic Program Generation*, International Series in Computer Science. Prentice-Hall.

Knoblock, T. and E. Ruf: 1996, 'Data Specialization'. In (PLDI'96, 1996), pp. 215–225, ACM SIGPLAN Notices, 31(5). Also TR MSR-TR-96-04, Microsoft Research, February 1996.

Kono, J. and T. Masuda: 2000, 'Efficient RMI: Dynamic Specialization of Object Serialization'. In: *Proceedings of the 20th International Conference on Distributed Computing Systems*. Taipei, Taiwan, pp. 308–315, IEEE Computer Society Press.

Lawall, J.: 1998, 'Faster Fourier Transforms via Automatic Program Specialization'. In (Hatcliff et al., 1998), pp. 338–355, Springer-Verlag.

Le Meur, A.-F., C. Consel, and B. Escrig: 2002, 'An Environment for Building Customizable Software Components'. In: *IFIP/ACM Conference on Component Deployment*. Berlin, Germany.

Lee, P. and M. Leone: 1996, 'Optimizing ML with Run-Time Code Generation'. In (PLDI'96, 1996), pp. 137–148, ACM SIGPLAN Notices, 31(5).

Marlet, R., S. Thibault, and C. Consel: 1999, 'Efficient Implementations of Software Architectures via Partial Evaluation'. *Journal of Automated Software Engineering* **6**(4), 411–440.

McCanne, S. and V. Jacobson: 1993, 'The BSD Packet Filter: A New Architecture for User-level Packet Capture'. In: *Proceedings of the Winter 1993 USENIX Conference*. San Diego, CA, USA, pp. 259–269, USENIX.

McNamee, D., J. Walpole, C. Pu, C. Cowan, C. Krasic, C. Goel, C. Consel, G. Muller, and R. Marlet: 2001, 'Specialization Tools and Techniques for Systematic Optimization of System Software'. *ACM Transactions on Computer Systems* **19**, 217–251.

Mock, M., C. Chambers, and S. J. Eggers: 2000, 'Calpa: a tool for automating selective dynamic compilation'. In: *International Symposium on Microarchitecture*. pp. 291–302.

Muller, G., R. Marlet, E. Volanschi, C. Consel, C. Pu, and A. Goel: 1998, 'Fast, Optimized Sun RPC Using Automatic Program Specialization'. In: *Proceedings of the 18th International Conference on Distributed Computing Systems*. Amsterdam, The Netherlands, IEEE Computer Society Press.

Muller, G., E. Volanschi, and R. Marlet: 1997, 'Scaling up Partial Evaluation for Optimizing the Sun Commercial RPC Protocol'. In: *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. pp. 116–125, ACM Press.

Muth, R., S. A. Watterson, and S. K. Debray: 2000, 'Code Specialization Based on Value Profiles'. In: *Static Analysis Symposium*. pp. 340–359.

PLDI'96: 1996, 'Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation'. Philadelphia, PA, USA: ACM SIGPLAN Notices, 31(5).

Plotkin, G.: 1975, 'Call-by-name, call-by-value, and the λ-calculus'. *Theoretical Computer Science* **1**(2), 125–159.

Ryder, B.: 1997, 'A Position Paper on Compile-time Program Analysis'. *ACM SIGPLAN Notices* **32**(1), 110–114.

Schultz, U., J. Lawall, C. Consel, and G. Muller: 1999, 'Towards Automatic Specialization of Java Programs'. In: *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, Vol. 1628 of *Lecture Notes in Computer Science*. Lisbon, Portugal, pp. 367–390.

Schultz, U. P., J. L. Lawall, and C. Consel: 2000, 'Specialization Patterns'. In: *Proceedings of the* 15^{th} *IEEE International Conference on Automated Software Engineering (ASE 2000)*. Grenoble, France, pp. 197–208, IEEE Computer Society Press.

Sun Microsystems: 1990, 'Network Programming Guide'. Sun Microsystems.

Taha, W. and T. Sheard: 1997, 'Multi-Stage Programming with Explicit Annotations'. In: *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-*

Based Program Manipulation. Amsterdam, The Netherlands, pp. 203–217, ACM Press.

Thibault, S. and C. Consel: 1997, 'A Framework for Application Generator Design'. In: *Proceedings of the Symposium on Software Reusability*. Boston, MA, USA.

Thibault, S., C. Consel, R. Marlet, G. Muller, and J. Lawall: 2000, 'Static and Dynamic Program Compilation by Interpreter Specialization'. *Higher-Order and Symbolic Computation* **13**(3), 161–178.

Thibault, S., C. Consel, and G. Muller: 1998, 'Safe and Efficient Active Network Programming'. In: *17th IEEE Symposium on Reliable Distributed Systems*. West Lafayette, Indiana.

Volanschi, E., C. Consel, G. Muller, and C. Cowan: 1997, 'Declarative Specialization of Object-Oriented Programs'. In: *OOPSLA'97 Conference Proceedings*. Atlanta, GA, USA, pp. 286–300, 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 degree of precision 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:

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

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 developer'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 $\mathcal{U} \sqsubseteq \mathcal{S} \sqsubseteq \mathcal{D}$. In addition to the ordinary least-upper-bound operation $\sqcup$, we define a commutative "strict least-upper bound" operator $\uplus$ such that, for any c:

$$c \uplus c = c \qquad \mathcal{U} \uplus \mathcal{S} = \mathcal{S} \qquad \mathcal{U} \uplus \mathcal{D} = \mathcal{D}$$

We also define an operation $c \oplus b$ that produces a binding time compatible with both the constraint c and the binding time b. This operation 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 $\uplus$ and $\oplus$ are not defined on all pairs of inputs, because some pairs of inputs such as $\mathcal{S} \uplus \mathcal{D}$ and $\mathcal{S} \oplus \mathsf{D}$ are incompatible. Thus these operations can be used both to compute a result 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 a binding-time annotated statement, and Γ' is a new binding-time environment reflecting 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-annotatedness rules are standard for a flow-sensitive binding-time analysis of an imperative language (Hornof and Noyé, 2000). We thus present only the rules that involve the constraints derived from the specialization module:

Variable Reference: The binding time of a variable reference `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 $*E$ takes into account the binding time b of the dereferenced expression E, as well as the binding times and constraints associated with the possible aliases of $*E$, *i.e.* the locations whose values might be accessed by the expression $*E$. The binding-time analysis must take into account the constraints on these locations, as well as the constraints on any parameters through which these locations are passed by reference.

The analysis distinguishes between several types of aliases, which we explain using the example shown in Figure 18. In the definition of `h`, the call `f(&a)` causes the parameter `x` to be bound to the address of the global variable `a`. Thus, we say that `a` is an alias of `*x`. Because `a` represents the location that contains the value, we further classify `a` as an *initial alias.* The binding time of `*x` must certainly respect the constraints on the initial alias `a`. But a principle of our approach is that all constraints are checked on function calls, to ensure that information is propagating through the program as requested by the program developer. Thus, we take into account the constraint on `*x` as well. Because no new memory location is allocated for the dereference of a parameter, the alias `*x` is referred to as an *intermediate alias.* Within the body of function `f` there is an assignment `c = x`. After this assignment, `*c` has two aliases: the initial alias `a` and the intermediate alias `*x`.

The example in Figure 18 also shows that a dereference expression can have multiple initial and intermediate aliases. Consider the function `g`. This function assigns the integer pointer variable `d` to either the parameter `z` or the global variable `c`. The single call site of `g`, in the definition of `f`, shows that `z` has as an initial alias the local variable `b`

```
int^S *^S c;
int^S a;
void h() {
  a = 4;
  f(&a);
}
```

```
void f(int^S *^S x) {
  int^U b = 3;
  c = x;
  ... g(&b) ...;
}
```

```
int g(int^D *^D z) {
  int^U *^U d;
  if(...) d = z;
  else d = c;
  return *d;
}
```

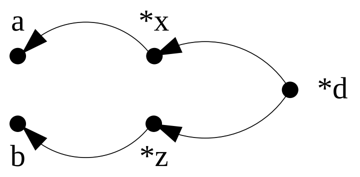


Figure 18. Initial and intermediate aliases

of function f. Thus, the assignment `d = z` implies that `*d` has initial alias `b` and intermediate alias `*z`. By our analysis of the value of `c` above, the assignment `d = c` implies that `*d` has initial alias `a` and intermediate alias `*x`. Because the alias analysis is not able to determine which branch of a conditional is taken, the complete set of aliases at the reference to `*d` at the end of `g` has both `a` and `b` as initial aliases, and `*x` and `*z` as intermediate aliases.

Having identified the kinds of aliases that can be associated with a dereference expression, we next consider how this information is taken into account to determine the binding time of the expression. Initial aliases refer to actual distinct memory locations, containing the possible values. Thus, we require that all constraints on the initial aliases be either $\mathcal{U}$ or identical, and combine these constraints using the strict combining operator $\uplus$. Intermediate aliases simply reflect parameter passing. These are combined with the least-upper-bound operator $\sqcup$, implying *e.g.* a dynamic constraint if any of the constraints is dynamic. The constraints on the initial aliases are combined with the constraints on the intermediate constraints using the strict $\uplus$ operator, to ensure that the constraints on the initial aliases are always respected. These constraints must furthermore be compatible with the inferred binding time for the dereference expression, which takes into account the binding times of the initial aliases. The complete rule for a dereference expression is thus as follows, where the operator $\mathcal{L}(*E)$ accesses the initial aliases κ and the intermediate aliases δ of the expression $*E$.

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

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

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

If there are multiple initial 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.

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:

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

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, the constraint on the parameter is checked to be compatible with the constraints on initial and intermediate aliases, and with the binding times of the initial 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 ($\mathtt{*}^i$ refers to i dereferences of `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.

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 as an immediate subexpression of a dynamic operation.[4] Both of these adjustments require the backwards propagation of information, and are performed by evaluation-time analysis (Hornof and Noyé, 2000).

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

The reannotation of a static expression having a non-liftable value as dynamic when the expression occurs as the immediate subexpression of a dynamic operation can constrain the propagation of static information during specialization. Consider the following example:

$\mathtt{int}^{\mathcal{D}}\ \mathtt{*}^{\mathcal{S}}\ \mathtt{x;}$

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

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

The declaration for `x` indicates that the result of the expression `*(x+1)` is dynamic. Thus, the static pointer-typed expression `(x+1)` is the immediate subexpression of a dynamic operation, and must itself be considered dynamic. This reannotation in turn implies that `x` is considered dynamic, in contradiction to the declaration $*^S$ 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.

To formalize the effect of specialization on locations declared to be static, the discussion in Section 5.2 uses an argument based on a semantics that makes explicit whether a subexpression occurs in a static or dynamic context. We now define the terms static context and dynamic context more precisely, and give the complete definition of this semantics.

A static context is one where simplification can necessarily occur if the context is filled with a static expression. For example, `if` ([]) `then` S_1 `else` S_2 is a static context because regardless of the value placed in the hole [], the `if` statement can be reduced to one of its branches. A context that does not have this property is a dynamic context. For example, [] `+ x` is a dynamic context because placing a value in the hole is not sufficient to make it possible to perform the addition computation. The precise definition of static and dynamic contexts for a language depends on what the associated partial evaluator considers to be a non-liftable value. As an example, we consider a partial evaluator for which integers are liftable and pointers are non-liftable.

Static and dynamic contexts are defined in terms of the target language of specialization. In the case of the source language presented at the beginning of the appendix, a typical partial evaluator would generate code in a target language containing all of the constructs of the source language, as well as constants and zero-argument functions. A constant results from replacing a static expression by its value, while a zero-argument function results from propagating the value of a static argument to the function body. Additionally, to better illustrate static and dynamic contexts, we assume that the source language contains addition and equality expressions, and thus include these expressions in the target language. The target language is thus as follows:

$$
\begin{array}{rlrcl}
G & \in & \text{Global} & ::= & T\,\texttt{x} \\
F & \in & \text{Function} & ::= & \texttt{f}\,\texttt{(}T\,\texttt{x)}\,S \\
T & \in & \text{Type} & ::= & \texttt{int} \mid T\,\texttt{*} \\
S & \in & \text{Statement} & ::= & L\,\texttt{=}\,E\texttt{;} \mid \texttt{\{}T\,\texttt{x; }S\texttt{\}} \mid \texttt{\{}S_1\,S_2\texttt{\}} \\
 & & & & \mid \texttt{if (}E\texttt{) then}\,S_1\,\texttt{else}\,S_2 \mid \texttt{f(}E\texttt{);} \mid \texttt{f();} \\
E & \in & \text{Expression} & ::= & \texttt{c} \mid \texttt{x} \mid \texttt{\&x} \mid \texttt{*}E \mid E_1\ \texttt{+}\ E_2 \mid E_1\ \texttt{==}\ E_2 \\
L & \in & \text{L-expression} & ::= & \texttt{x} \mid \texttt{*}E
\end{array}
$$

For the statements and expressions in this language, static contexts are defined as follows, where S and E are target-language terms, as defined above:

$$
\begin{array}{rlrcl}
C_S & \in & \text{Static statement context} & ::= & L\,\texttt{=}\,C_E\,\texttt{;} \\
 & & & & \mid \texttt{\{}T\,\texttt{x; }C_S\texttt{\}} \mid \texttt{\{}C_S\,S\texttt{\}} \mid \texttt{\{}S\,C_S\texttt{\}} \\
 & & & & \mid \texttt{if (}[\,]\texttt{) then}\,S_1\,\texttt{else}\,S_2 \\
 & & & & \mid \texttt{if (}C_E\texttt{) then}\,S_1\,\texttt{else}\,S_2 \\
 & & & & \mid \texttt{if (}E\texttt{) then}\,C_S\,\texttt{else}\,S \\
 & & & & \mid \texttt{if (}E\texttt{) then}\,S\,\texttt{else}\,C_S \\
 & & & & \mid \texttt{f(}C_E\texttt{);} \\
 & & & & \\
C_E & \in & \text{Static expression context} & ::= & \texttt{*}C_E \mid C_E\ \texttt{+}\ E \mid E\ \texttt{+}\ C_E \\
 & & & & \mid [\,]\ \texttt{==}\ c \mid c\ \texttt{==}\ [\,] \\
 & & & & \mid C_E\ \texttt{==}\ E \mid E\ \texttt{==}\ C_E
\end{array}
$$

In the last two cases of the definition of a static expression context, E is assumed not to be a constant expression.

These definitions are motivated as follows. A conditional statement can always be reduced when the test expression is static, regardless of whether the value of the test expression is liftable or non-liftable (we consider a C-like language, in which 0 represents false and all non-zero values are considered to be true). Because the result of a comparison is an integer, a comparison against a constant (possibly the result of specializing a static expression) can always be reduced when the compared expression is static. On the other hand, the result of an addition involving a non-liftable value in our language is always a non-liftable value, and so neither argument of an addition expression can be the hole of a static context. A dynamic context is simply a context whose hole is in the position of an expression and that is not a static context.

To describe the relationship between the effect of specialization and the specialization module declarations, we use a semantics that keeps track of whether each evaluated expression is in a possible position of the hole of a static context (indicated by a judgment of the form

$\mathsf{S}, \rho \models_{\mathrm{e}} E : v$) or of a dynamic context (indicated by a judgment of the form $\mathsf{D}, \rho \models_{\mathrm{e}} E : v$). Extracts of this semantics have been presented in the body of the paper. The complete semantics of statements and expression is shown below. The semantics uses an initial environment ρ_0 containing the global variables.

Statements:

$$\frac{\rho \models_{\mathrm{l}} L : \ell \qquad \mathsf{D}, \rho \models_{\mathrm{e}} E : v}{\rho, \Phi \models_{\mathrm{s}} L\,\texttt{=}\,E\texttt{;} : \rho[\ell \mapsto v]} \qquad \frac{\rho[\mathtt{x}_\ell \mapsto \bot], \Phi \models_{\mathrm{s}} S : \rho'[\mathtt{x}_\ell \mapsto v]}{\rho, \Phi \models_{\mathrm{s}} \texttt{\{}T\,\mathtt{x}\texttt{;}\ S\texttt{\}} : \rho'}$$

$$\frac{\rho, \Phi \models_{\mathrm{s}} S_1 : \rho_1 \qquad \rho_1, \Phi \models_{\mathrm{s}} S_2 : \rho_2}{\rho, \Phi \models_{\mathrm{s}} \texttt{\{}S_1\, S_2\texttt{\}} : \rho_2} \qquad \rho, \Phi \models_{\mathrm{s}} \texttt{\{\}} : \rho$$

$$\frac{\mathsf{S}, \rho \models_{\mathrm{e}} E : v \qquad v \neq 0 \qquad \rho, \Phi \models_{\mathrm{s}} S_1 : \rho'}{\rho, \Phi \models_{\mathrm{s}} \texttt{if (}E\texttt{) then}\, S_1\, \texttt{else}\, S_2 : S_1 : \rho'}$$

$$\frac{\mathsf{S}, \rho \models_{\mathrm{e}} E : 0 \qquad \rho, \Phi \models_{\mathrm{s}} S_2 : \rho'}{\rho, \Phi \models_{\mathrm{s}} \texttt{if (}E\texttt{) then}\, S_1\, \texttt{else}\, S_2 : S_2 : \rho'}$$

$$\frac{\mathsf{D}, \rho \models_{\mathrm{e}} E : v \qquad \mathtt{f} \mapsto (\mathtt{x}, S) \in \Phi \qquad \rho_0[\mathtt{x}_\ell \mapsto v], \Phi \models_{\mathrm{s}} S : \rho'[\mathtt{x}_\ell \mapsto v']}{\rho, \Phi \models_{\mathrm{s}} \mathtt{f}\texttt{(}E\texttt{)} : \rho'}$$

$$\frac{\mathtt{f} \mapsto (-, S) \in \Phi \qquad \rho_0, \Phi \models_{\mathrm{s}} S : \rho'[\mathtt{x}_\ell \mapsto v']}{\rho, \Phi \models_{\mathrm{s}} \mathtt{f}\texttt{(}E\texttt{)} : \rho'}$$

Expressions:

$$b, \rho \models_{\mathrm{e}} \mathtt{c} : c \qquad \frac{b, \mathtt{x}_\ell \mapsto v \in \rho}{b, \rho \models_{\mathrm{e}} \mathtt{x} : v} \qquad b, \rho \models_{\mathrm{e}} \texttt{\&x} : \mathtt{x}_\ell$$

$$\frac{\mathsf{D}, \rho \models_{\mathrm{e}} E : \ell \qquad b, \ell \mapsto v \in \rho}{b, \rho \models_{\mathrm{e}} \texttt{*}E : v} \qquad \frac{\mathsf{D}, \rho \models_{\mathrm{e}} E_1 : v_1 \qquad \mathsf{D}, \rho \models_{\mathrm{e}} E_2 : v_2}{b, \rho \models_{\mathrm{e}} E_1 \texttt{ + } E_2 : v_1 + v_2}$$

$$\frac{\mathsf{S}, \rho \models_{\mathrm{e}} E_1 : v_1}{b, \rho \models_{\mathrm{e}} E_1 \texttt{ == } \mathtt{c} : v_1 = c} \qquad \frac{\mathsf{S}, \rho \models_{\mathrm{e}} E_2 : v_2}{b, \rho \models_{\mathrm{e}} \mathtt{c} \texttt{ == } E_2 : c = v_2}$$

$$\frac{\mathsf{D}, \rho \models_{\mathrm{e}} E_1 : v_1 \qquad \mathsf{D}, \rho \models_{\mathrm{e}} E_2 : v_2}{b, \rho \models_{\mathrm{e}} E_1 \texttt{ == } E_2 : v_1 = v_2} (E_1 \text{ and } E_2 \text{ are not constants})$$

L-values:

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