THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TR11/06

Request and Assert: A pragmatic approach to generating specialization scenarios

Ping ZHU and Siau-Cheng KHOO

November 2006

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

Request and Assert: A pragmatic approach to generating specialization scenarios

Ping Zhu and Siau-Cheng Khoo

Department of Computer Science
National University of Singapore

November 13, 2006

Abstract

A specialization scenario provides a programmer friendly mechanism communicating the information about specialization opportunities to partial evaluators. Unfortunately, the process of generating suitable scenarios remains an art only mastered by programmers with in-depth knowledge about partial evaluation. Existing works on generating scenarios either rely on a brute-force approach to generate all possible scenarios, or to introduce specific design patterns into the programming to facilitate extracting specialization scenarios. In this paper, we provide a lightweight approach to partial evaluation by enabling non-experts to declare two simple specialization concerns: *request* and *assert*. The *request* enables a programmer to declare specialization opportunities and an *assert* aims to prevent undesirable partial evaluation, such as infinite specialization, from occurring. We describe an algorithm that derives specialization scenarios by declaring the necessary binding-time values at program inputs, aiming at fulfilling any request and satisfy the assert meanwhile.

1 Introduction

Partial evaluation [12] is an automatic program specialization technique that specializes a program by aggressively propagating some constants provided in a given usage context. A partial evaluator reduces *static* expressions or statements, which depend only on information that can be inferred from the constants and static control flows, and reconstructs those expressions or statements which rely on *dynamic* information. In the case of *offline* partial evaluation, which will be the focus of this paper, the partial evaluation typically involves two phases: a preprocessing phase called *binding-time analysis* that attempts to determine the static computations at each program point, followed by a *specialization* phase that constructs the specialized program, based on the binding-time information derived from the preprocessing phase. This basic approach has been extended with numerous advanced features ([3, 6, 7, 9], etc.), thus making partial evaluation scale up to the realistic language and play significant role in program specialization. Many practical partial evaluators ([13, 5, 14]) have been implemented.

Despite these extensions and advances, partial evaluation remains an art only mastered by programmers with in-depth knowledge about partial evaluation. As explained in [14],

> " ... one reason seems to be the amount of work needed to identify specialization opportunities in the source code and to ensure that the partial evaluator can take

> advantage of these opportunities. Even in the case when code fragments have been identified that should benefit from partial evaluation, there exists little support for communicating information about specialization opportunities to partial evaluators."

In response to the demand for a programmer-friendly partial evaluation, Le Meur *et al.* [14] propose *specialization scenario* as a declaration that identifies a function or global variable or data structure, which are termed as *specialization parameters in the literature*, those are of interest for specialization and declares the binding-time context in which specialization involving these constructs should be carried out.

The specialization scenario has been widely used in many partial evaluation applications. In [2], Bobeff *et al.* proposed a to systematically creates all possible and consistent specialization scenarios for a component and then allow component developers to intervene by manually removing those unsuitable specialization contexts by considering benefits in terms of specialization opportunities. Despite its being declarative in nature, the process required in deriving a specialization scenario remains a challenging task, and requires programmers' much insight into the mechanics and strategy of partial evaluation. In [15], Schultz *et al.* proposes *specialization patterns* approach, which describes how to capture specialization opportunities provided by specific uses of design patterns. But this approach also introduces extra overhead of design patterns into plain programming.

We deem the limitation of specialization scenario approach [14] as: it only allows programmers to express specialization intentions in terms of external specialization parameters. It does not provide programmers convenient means to express their specialization intensions internally by taking advantage of their knowledge about the program code. It is true that PE is meant to take away something that is known during compilation time. But this does not mean that "something known during compilation time" must be obtained from the program input. It may be discovered from the nature of the code itself. However, we still value the way of defining specialization context for a component in terms of component's parameters because it is friendly for inter-component specialization and it also provides component users some indication or guidance in component

specialization.

In this paper, we propose a lightweight and declarative approach to derive specialization scenarios. Our approach relies on two simple specialization concerns: *Request* and *Assert*. The role of a *request* is to identify a code segment s as a specialization opportunity. Semantically, this request indicates the programmer's desire to make the binding-time value of s static. From these requests we then direct our binding-time computation to fulfil them. The generated specialization scenario thus aims to warrant such a request by declaring the necessary binding-time values at program inputs. We call a specialization *profitable* when it can lead to a fulfilment of any such request.

While we allow the programmer to set `Request` at any code segment, we find that in most situations, the profitable specialization is derived from the elimination of conditional test at partial evaluation time. Hence, in our attempt towards automatic generation of specialization scenarios, we set `Request` to occur at each conditional test in function definitions. Thus, for the rest of the paper, we take the position that *profitable specialization of a function definition can be automatically derived from either direct or indirect elimination of conditional tests.* (Indirect elimination refers to the elimination of conditional tests that occur at other function definitions, called by the current one.)

Consider the following example:

```
int add(int x, int y)
{   return x+y;    }

int mul(int x, int y) {
  int z;
  if x > 0 {                        // test1
    z = mul(x-1,y);
    return add(z,y); }
  if x == 0                         // test2
    return 0;
  if y > 0 {                        // test3
    z = mul(x,y-1);
    return add(z,x); }
  return mul (-x, -y);
}
```

Upon the request that any of the conditional tests in `mul` be made static, our approach derives the following specialization scenarios:

$$\begin{array}{ll} \mathtt{bt_x} = \mathtt{S} \wedge \mathtt{bt_y} = \mathtt{D} & \text{(Scenario 1)} \\ \mathtt{bt_x} = \mathtt{D} \wedge \mathtt{bt_y} = \mathtt{S} & \text{(Scenario 2)} \\ \mathtt{bt_x} = \mathtt{S} \wedge \mathtt{bt_y} = \mathtt{S} & \text{(Scenario 3)} \end{array}$$

Scenario 1 makes both test1 and test2 static; scenario 2 makes test3 static; scenario 3 makes those three conditional tests static.

Note that specialization scenarios are not generated by typical forward-fashion binding-time analysis. Instead, they are generated by propagating outwardly binding-time request at the program point possessing a specialization opportunity.

While a *request* enables programmer to declare a specialization opportunity, an *assert* aims to prevent undesirable partial evaluations, such as infinite specialization, from occurring. Such a control of the partial evaluation is achieved by placing constraints over the possible binding-time values of program inputs. Consequently, we can view specialization scenarios thus produced as a constraint-based scenarios, analogous to the constraint-based type declaration.

The main contribution of this paper are as follows:

- We provide a declarative mechanism that enables non-experts to pin-point places in the program as certain specialization opportunities to be addressed in the name of request,
- We provide a lightweight mechanism that enables non-experts to control part of the partial evaluation by setting a binding-time constraint (i.e. *assert* over variable.
- We propose an automatic analysis to generate specialization scenarios which fulfill the request and assert meanwhile.

The rest of this paper is organized as follows. Section 2 introduces the core language and some notational conventions used for presenting our approach. We elaborate the idea of using *request*

and *assert* to control partial evaluation in Section 3. In Section 4 we describe how to automatically generate the specialization scenarios which fulfil the *request* and *assert* declared by programmers inside a function definition. Section 5 completes the picture by describing how specialized code can be generated. We conclude in Section 6.

2 Preliminaries

The core langauge used in this paper is a subset of C language, whose abstract syntax is defined in Figure 1.

v	$\in$	**Var**	Variables
f, g	$\in$	**FName**	Function names
n	$\in$	$\mathcal{R}$	Real numbers
b_{op}	$\in$	**BOp**	Binary operators
	::=	$+ \mid - \mid * \mid / \mid == \mid != \mid < \mid > \mid >= \mid <= \mid \&\& \mid \mid\mid$	
e	$\in$	**Exp**	Expressions
	::=	$n \mid v \mid f\ (e_1, \ldots, e_n) \mid e_1\ b_{op}\ e_2$	
s	$\in$	**Stat**	Statements
	::=	$s_1; s_2 \mid \texttt{while}\ e\ s \mid \texttt{return}\ e \mid v = e \mid \texttt{if}\ e\ s_1\ \texttt{else}\ s_2 \mid$	
$decl$	$\in$	**Decl**	Declarations
	::=	$\texttt{int}\ v$	
fd	$\in$	**FDef**	Function definitions
	::=	$\texttt{int}\ f\ (decl^+)\ \{decl^*\ ;\ s\}$	

Figure 1: Syntax of Core Language - C subset

The core language excludes features such as pointers, compound data structures, global variables. The evaluation strategy of function call is limited to call-by-value and every function definition must return a value.

The notational convention for expressing binding-time information is listed as following. The term *binding-time* is sometimes abbreviated as BT for the convenience of presentation.

bt_v	$\in$	$\mathbf{BT_v}$	BT variables
bt_e	$\in$	$\mathbf{BT_e}$	BT expressions
	::=	$\texttt{S} \mid \texttt{D} \mid bt_v \mid bt_{e_1} \sqcup bt_{e_2} \mid bt_{e_1} \sqcap bt_{e_2}$	

There are three primitive binding-time expressions: two binding-time constants `S` and `D` representing a static value and a dynamic value respectively; and a binding-time variable bt_v ranging over `S` and `D`. A complex binding-time expression is formed using two operators: least upper bound $\sqcup$ and greatest lower bound $\sqcap$. The two binding-time constants `S` and `D` are ordered in decreasing staticness: $\texttt{S} \sqsubset \texttt{D}$. This ordering can be naturally extended to partial ordering over tuples of binding-time values. *The specialization scenario with respect to a function definition is a conjunctive of equivalences of the form* $\bigwedge(bt_{v_i} = \texttt{S}|\texttt{D})$ *where* bt_{v_i} *stand for the binding-time variables pertaining to function's parameters.*

3 Two Specialization Concerns: Request and Assert

The idea of using specialization scenarios in dictating partial evaluation process has provided a great leap towards making partial evaluator acceptable to non-experts. Unfortunately, the process of generating suitable scenarios remains an art only mastered by programmers with in-depth knowledge about partial evaluation. We propose a lightweight and declarative approach, which relies on two simple specialization concerns: *Request* and *Assert*, to derive specialization scenarios.

3.1 Declaring Profitable Specialization: Request

In this paper, we propose a new mechanism that enables the programmer to identify points of specialization opportunities within a function definition by enclosing a code segment e with a construct `Request`, as in (`Request` e). Semantically, the construct `Request` indicates user's desire to make the binding-time value of e static. Note that we can also make request over function parameters directly such that our approach is a generalization of the approach presented in [14].

We then employ a series of analyses to derive a specialization scenario which declares the appropriate binding-time value over the function's parameters in order to realize the requested specialization opportunities. Our main thesis is that *specialization scenarios are produced to ensure the fulfillment of requests*

Consider the following two function definitions, in which two requests for profitable specialization have been declared.

```
int m(int x, int y) {
  if Request (x == 0)
    return y+1;
  else return y-1;
}
```

```
int n(int x, int y) {
  int i;
  i = m(x, y);
  if Request (i == 0)
    return 1;
  else return 2;
}
```

For function `m`, the programmer wishes for the binding-time of the conditional test `(x = 0)` to be static. This implies that the binding-time of input `x` should be static, whereas the binding-time of `y` is not restricted. For function `n`, the programmer wishes for the binding-time of the conditional test `(i = 0)` to be static. This can be achieved only when the call to `m` has a static return value. As shown by the definition of function `m` above, this in turns requires both parameters of `n` to be static.

Combining both results, we conclude that specialization of function `n` can be profitable in two ways: (1) when the input `x` is static, which results in the test in `m` to be specialized away; and (2) when both inputs `x` and `y` are static, which results in all tests in both `m` and `n` to be specialized away. For the `m` and `n` function definition, we have the following scenarios:

$$\begin{array}{ll} \texttt{m} : \texttt{bt}_\texttt{x} = \texttt{S} \wedge \texttt{bt}_\texttt{y} = \texttt{D} & \text{(Scenario 1 for m)} \\ \texttt{m} : \texttt{bt}_\texttt{x} = \texttt{S} \wedge \texttt{bt}_\texttt{y} = \texttt{S} & \text{(Scenario 2 for m)} \\ \texttt{n} : \texttt{bt}_\texttt{x} = \texttt{S} \wedge \texttt{bt}_\texttt{y} = \texttt{D} & \text{(Scenario 1 for n)} \\ \texttt{n} : \texttt{bt}_\texttt{x} = \texttt{S} \wedge \texttt{bt}_\texttt{y} = \texttt{S} & \text{(Scenario 2 for n)} \end{array}$$

Except for the scenario 2 of function `m`, all the scenarios are created to fulfil at least one `Request`. Scenario 2 of function `m` is created to propagated static computation through function `m`, so that static computation at function `n` does not get obstructed.

A function definition without any `Request` is one that does not have any conditional test, thus

a plain linear code (assuming the function always terminate). In this case, the corresponding specialization context associated with this function definition is derived from its calling contexts.

3.2 Controlling Infinite Specialization: Assert

By requesting that some conditional tests be static at each specialization scenario, we can sometimes prevent infinite specialization from arising. For the following example,

```
int mb(int x, int y) {
  if x > 0 return mb(x-1,y-1);
  else return y;
}
```

By requesting that the conditional test `x > 0` be static, the specialization scenario obtained from `mb` is $\texttt{bt}_{\texttt{x}} = \texttt{S}$; that is, function `mb` will be effectively specialized when `x` is static. In this case when `mb` is called with `x` is dynamic and `y` static, we can*not* find a specialization scenario of `mb` that covers this call context; hence, we residual the call; ie., specializing the call by raising the binding-time of `y` to dynamic.

Unfortunately, using `Request` to prevent infinite specialization is not always appropriate, and there are cases when infinite specialization cannot be avoided. Take the following contrived example:

```
int mc(int x, int y) {
  if x > 0                          // test1
    return mc(x-1,y-1)
  else if y > 0                     // test2
    return mc(x-1,y-1)
  else return y
}
```

Under the two requests that test1 and test2 be made static, `mc` will have two specialization scenarios:

$\texttt{bt}_{\texttt{x}} = \texttt{S} \wedge \texttt{bt}_{\texttt{y}} = \texttt{D}$ (Scenario 1 : test1 is made static)
$\texttt{bt}_{\texttt{x}} = \texttt{S} \wedge \texttt{bt}_{\texttt{y}} = \texttt{S}$ (Scenario 2 : test2 is made static)

However, consider the case when `mc` is called in a specialization context with $\texttt{x} = 1$ and `y` being unknown. This context matches the scenario $\texttt{bt}_\texttt{x} = \texttt{S} \wedge \texttt{bt}_\texttt{y} = \texttt{D}$. Specializing `mc` with this context will however place the second recursive call under dynamic control, leading to infinite specialization. A snapshot of the specialization is shown below:

mc_1(y) {	mc_0(y) {	mc_{-1}(y) {	....
return mc_0(y-1);	if y > 0	if y > 0	
}	return mc_{-1}(y-1);	return mc_{-2}(y-1;z);	
	else return y;	else return y;	
	}	}	

In this paper, we introduce a notion of *assert* to curb such infinite specialization from arising. The syntax of the assert ξ is as follows, where bt_e represents the binding-time expression we defined in Section 2:

$$\begin{array}{lll} \xi & \in & \mathbf{BT_c} \\ & ::= & bt_{e_1}\ op\ bt_{e_2} \mid \xi_1 \wedge \xi_2 \mid \xi_1 \vee \xi_2 \\ op & ::= & \leq \mid \texttt{=} \end{array}$$

The objective of an assert is to declare a (set of) constraint over the binding-time values of variables used at a particular program point. For a specialization to take place, the related assert must be met. In our approach, since specialization is controlled by specialization scenarios, *an assert can effectively eliminate some unsafe scenarios.*

The use of binding-time constraint in asserts reflects our desire to control specialization effectiveness in a declarative fashion. For the above `mc` example, we can provide the following assert at the function header:

$$\texttt{mc(x,y;z)} \quad \texttt{Assert}\ \texttt{bt}_\texttt{x} = \texttt{bt}_\texttt{y}$$

This asserts that the binding-time for both input parameters of `mc` should always be the same when specializing the function. This effectively rules out the earlier two scenarios of `mc`: $\texttt{bt}_\texttt{x} = \texttt{S} \wedge \texttt{bt}_\texttt{y} = \texttt{D}$ and $\texttt{bt}_\texttt{x} = \texttt{D} \wedge \texttt{bt}_\texttt{y} = \texttt{S}$. Consequently, function can only be specialized when both the input are static (obtained from the resolution of the constraint $(\texttt{bt}_\texttt{x} = \texttt{S} \wedge \texttt{bt}_\texttt{y} = \texttt{D}) \vee (\texttt{bt}_\texttt{x} =$

$\mathtt{D} \wedge \mathtt{bt_y} = \mathtt{S}) \wedge (\mathtt{bt_x} = \mathtt{bt_y}))$.

A more refined use of assert is possible: placing different assert at different function calls occurring in the function body. Following the `mc` example, we provide asserts at program points `2` and `4` respectively.

```
int mc(int x, int y)
   2: Assert bt_x ⊑ bt_y
   4: Assert bt_y ⊑ bt_x
{
   1: if x > 0
   2:     return mc(x-1,y-1)
   3: else if y > 0
   4:     return mc(x-1,y-1)
   5: else return y
}
```

The above asserts state that in the first call, the second input should be at least as static as the first input; in the second call, the first input should be at least as static as the second input. Hence, given the specialization context of $\mathtt{x} = 1$ and `y` being unknown, we will allow the first call to be specialized aggressively, but the second call will not be specialized against any arguments:

```
mc_1(y)                    mc_0(y)
{                          {
   return mc_0(y-1);          if y > 0
}                             return mc(x-1,y-1);
                              else return y;
                           }
```

Effectively, these asserts captures the decision of poor man's generalization on function parameters, as described in [10, 4].

In our effort towards automatic generation of specialization scenarios, we can choose to insert assert at each calls appearing under conditional control flow, capturing the essence of poor man's generalization. Certainly, this does not always guarantee termination of partial evaluation, and more recent research into termination analysis should be employed, and the result be expressed in

terms of asserts.

3.3 Program Constructs for Specialization Concerns Annotations

We provide two program constructs for the programmers to declare request and assert in the source program:

- A request is represented by `Request` e, where e is an expression which are identified as specialization opportunities.
- An assert, associated with a program point for a statement, is represented as `Assert` (ξ), where ξ is a disjunction of binding-time constraints over the arguments of function call within that statement.

Figure 2 demonstrates an example of specialization concerns annotation:

```
int mc(int x, int y)
   2: Assert  bt(x−1) ⊑ bt(y−1)
   4: Assert  bt(y−1) ⊑ bt(x−1)
{
   1: if Request (x > 0)
   2:     return mc(x-1,y-1)
   3: else if Request (y > 0)
   4:     return mc(x-1,y-1)
   5: else return y
}
```

Figure 2: Specialization Concerns Annotation

The set of programmer-desired specialization scenarios produced w.r.t these annotations should possess the properties that: *every single specialization scenario should fulfill at least one piece of request primitive, meanwhile, it should satisfy all the assert primitives* ξ

4 Computing Specialization Scenarios

We have developed and implemented a modular analysis to compute the profitable specialization context for each function definition. There are two steps in the analysis. In the first step we perform a dependence analysis over a set of interrelated function definitions, and returns at each program point, a local binding-time environment $\rho \in \mathbf{BT_{env}}$, of the form $[v \mapsto bte]$. v ranges over program variables occurring at the program point; bte is a binding-time expression of v.

A global function name indexed table $\tau \in \mathbf{F_{tab}}$, of the form $[f \mapsto (ps,\ ret)]$ is needed for the interprocedual analysis, where ps is the parameter list of the function f; ret is a binding-time expression of f's return value in terms of the binding-time variables associated with f's parameters.

Then we generate binding-time constraints for request/assert annotations. The binding-time constraints are expressed in terms of the binding-time expressions of the variables stored in associated binding-time environment.

Compared with the existing approaches in generating specialization scenarios, such as [14] and [2], our approach fills the gap between expressing conceptual specialization intentions and generating specialization scenarios which are necessary for offline partial evaluator.

4.1 Dependence Analysis

Dependence analysis aims to compute, at each program point, *the binding-time expressions of the variables occurring at the program point, which are expressed in terms of the binding-time expression of function parameters.* The analysis takes into consideration both dataflow and control flow information, which is in line with customary treatment in binding-time analysis whose safety is validated in [8]. The result of dependence analysis can be used in computing different specialization scenarios provided with different request/assert annotations, thus saving substantial computation cost. The complete dependence analysis is defined in Figures 3 and 4.

$$
\begin{array}{l}
\mathcal{BTA}_P :: \mathcal{P}(\mathbf{FDef}) \rightarrow \mathbf{F_{tab}} \\
\quad let \;\; \mathcal{BTA}'_P\ (fd_1, \ldots, fd_n)\ \tau_1 \;=\; let \quad \tau_{i+1} \;=\; \mathcal{BTA}_f\ fd_i\ \tau_i \quad \forall\, i \in \{1, \ldots, n\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad in \quad if\ \tau_{i+1} = \tau_1\ then\ \tau_{i+1} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad else\ \mathcal{BTA}'_P\ (fd_1, \ldots, fd_n)\ \tau_{i+1} \\
\qquad\quad \tau_0 \;=\; let \quad \{f_i \;=\; name\ (fd_i) \mid 1 \leq i \leq n\} \\
\qquad\qquad\qquad\quad \{ps_i \;=\; para\ (fd_i) \mid 1 \leq i \leq n\} \\
\qquad\qquad\quad in \quad \{f_i \mapsto (ps_i,\ \mathtt{S},\ btc_{f_i}) \mid 1 \leq i \leq n\} \\
\quad in \quad \mathcal{BTA}'_P\ (fd_1, \ldots, fd_n)\ \tau_0
\end{array}
$$

$$
\begin{array}{l}
\mathcal{BTA}_f :: \mathbf{FDef} \rightarrow \mathbf{F_{tab}} \rightarrow \mathbf{F_{tab}} \\
\mathcal{BTA}_f\ fd\ \tau = \\
\quad let \quad f = name\ (fd) \\
\qquad\quad [\![s]\!] = body\ (fd) \\
\qquad\quad ps = para\ (fd) \\
\qquad\quad \rho = InitBT_{env}(ps) \\
\qquad\quad ctr = \mathtt{S} \\
\qquad\quad (\_,\ \tau') = \mathcal{BTA}_s\ [\![s]\!]\ \rho\ \tau\ f\ ctr \\
\quad in \quad \tau'
\end{array}
$$

$$
InitBT_{env}(ps) = \{v_i \mapsto bt_{v_i} \mid v_i \in ps\}
$$

Figure 3: $\mathcal{BTA}_p$ and $\mathcal{BTA}_f$: Binding-time Analysis – Part 1

The main analysis function, $\mathcal{BTA}_p$ operates on a set of function definitions (fd_1, ..., fd_n), computes the least fix-point for environment $\mathbf{F_{tab}}$. At the beginning, the $\mathbf{F_{tab}}$ is initialized such that the binding-time value of each function's return value is static.

$\mathcal{BTA}_p$ is defined in terms of two sub-analysis: $\mathcal{BTA}_f$ which is a binding-time analysis over a single function definition; and $\mathcal{BTA}_s$ which operates on the function body. The function $InitBT_{env}(ps)$ used $\mathcal{BTA}_f$ initializes the binding-time environment at function entry point in the way that every parameter in ps is assigned with a binding-time variable.

$\mathcal{BTA}_s$ updates the binding-time environments with the change of data flow or control flow information. Note that, in the case of conditional expression, control flow information is captured at each branch of the conditional. $\mathcal{BTA}_s$ is relying on a function to computing binding-time value for expression, which is defined in Figure 5.

The operators $\bowtie$, $\uplus_\rho$ and $\uplus_\tau$ used in $\mathcal{BTA}_s$ are defined as:

- $\bowtie$ extends an environment with new entries.

$$\mathcal{BTA}_s :: \mathbf{Stat} \to \mathbf{BT_{env}} \to \mathbf{F_{tab}} \to \mathbf{FName} \to \mathbf{BT_e} \to (\mathbf{BT_{env}}, \mathbf{F_{tab}})$$

$$\begin{array}{l}
\mathcal{BTA}_s\ [\![v = e]\!]\ \rho\ \tau\ f\ ctr = \\
\quad let\ \ bt_v = (\mathcal{BTA}_e\ e\ \rho\ \tau) \sqcup ctr \\
\quad in\ \ (\rho[v \mapsto bt_v],\ \tau) \\
\\
\mathcal{BTA}_s\ [\![\texttt{int}\ v_1,\ \ldots,\ v_n]\!]\ \rho\ \tau\ f\ ctr = \\
\quad let\ \ \{bt_{v_i} = newBTVar(v_i) \mid 1 \le i \le n\} \\
\quad in\ \ (\rho \bowtie (\bigcup_{1 \le i \le n} \{v_i \mapsto bt_{v_i}\}),\ \tau) \\
\\
\mathcal{BTA}_s\ [\![\texttt{if}\ e\ s_1\ \texttt{else}\ s_2]\!]\ \rho\ \tau\ f\ ctr = \\
\quad let\ \ ctr' = ctr \sqcup (\mathcal{BTA}_e\ e\ \rho\ \tau) \\
\qquad (\rho_1,\ \tau_1) = \mathcal{BTA}_s[\![s_1]\!]\ \rho\ \tau\ f\ ctr' \\
\qquad (\rho_2,\ \tau_2) = \mathcal{BTA}_s[\![s_2]\!]\ \rho\ \tau\ f\ ctr' \\
\quad in\ \ (\rho_1 \uplus_\rho \rho_2,\ \tau_1 \uplus_\tau \tau_2) \\
\\
\mathcal{BTA}_s\ [\![\texttt{while}\ e\ s]\!]\ \rho\ \tau\ f\ ctr = \\
\quad let\ \ ctr' = ctr \sqcup (\mathcal{BTA}_e\ e\ \rho\ \tau) \\
\qquad (\rho',\ \tau') = \mathcal{BTA}_s[\![s]\!]\ \rho\ \tau\ f\ ctr \\
\quad in\ \ if\ \rho = \rho' \\
\qquad then\ (\rho',\ \tau') \\
\qquad else\ \mathcal{BTA}_s[\![s]\!]\ \rho'\ \tau'\ f\ ctr' \\
\\
\mathcal{BTA}_s\ [\![\texttt{return}\ e]\!]\ \rho\ \tau\ f\ ctr = \\
\quad let\ \ bt_{ret} = (\mathcal{BTA}_e\ e\ \rho\ \tau) \sqcup ctr \\
\quad in\ \ (\rho \bowtie \{ret \mapsto bt_{ret}\},\ \tau[f \mapsto (\tau(f).ps, \tau(f).ret \sqcup bt_{ret})]) \\
\\
\mathcal{BTA}_s\ [\![s_1;s_2]\!]\ \rho\ \tau\ f\ ctr = \\
\quad let\ \ (\rho_1,\ \tau_1) = \mathcal{BTA}_s\ [\![s_1]\!]\ \rho\ \tau\ f\ ctr \\
\quad in\ \ \mathcal{BTA}_s\ [\![s_2]\!]\ \rho_1\ \tau_1\ f\ ctr
\end{array}$$

Figure 4: Binding-time Analysis – Part 2

- $\rho_1 \uplus_\rho \rho_2 = \{x \mapsto bte_1 \sqcup bte_2 | x \mapsto bte_1 \in \rho_1,\ x \mapsto bte_2 \in \rho_2\}$
- $\tau_1 \uplus_\tau \tau_2 = \{f \mapsto (ps, ret_1 \sqcup ret_2) | f \mapsto (ps, ret_1) \in \tau_1,\ f \mapsto (ps, ret_2) \in \tau_2\}$.

4.2 Correctness

The correctness of dependence analysis is intimately related to the idea of congruence divisions as originally proposed by Jones [11]. Basically, the return value depends on a set of function's parameters if both the set of parameters and the return value fall within the same static partition.

$$\mathcal{BTA}_e :: \mathbf{Exp} \rightarrow \mathbf{BT_{env}} \rightarrow \mathbf{F_{tab}} \rightarrow \mathbf{BT_e}$$

$$\mathcal{BTA}_e\ [\![n]\!]\ \rho\ \tau = \mathtt{S}$$

$$\mathcal{BTA}_e\ [\![v]\!]\ \rho\ \tau = \rho(v)$$

$$\mathcal{BTA}_e\ [\![e_1\ b_{op}\ e_2)]\!]\ \rho\ \tau = (\mathcal{BTA}_e[\![e_1]\!]\ \ \rho\ \tau) \sqcup (\mathcal{BTA}_e[\![e_2]\!]\ \ \rho\ \tau)$$

$$\begin{array}{lll} \mathcal{BTA}_e\ [\![f\ (e_1,\ldots,e_n)]\!]\ \rho\ \tau = & let & \{bt_{e_i} = \mathcal{BT}_e[\![e_i]\!]\ \ \rho\ \tau \mid 1 \leq i \leq n\} \\ & & (v_1,\ldots,v_n) = \tau(f).ps \\ & in & \tau(f).ret[v_1 \mapsto bt_{e_1},\ldots,v_n \mapsto bt_{e_n}] \end{array}$$

Figure 5: Computing Binding-time Expressions for Expressions

That is, the staticness of the set of parameters ensures static computation of the return value.

Equivalently, we can define the correctness of our dependence analysis in terms of standard evaluation of function invocation under two slightly different assignments of input parameters. To begin with, we formally define what we mean by an assignment, and a complete extension of an assignment.

Definition 1 *Given a set of variables V, an* assignment *A to V is a mapping of each variable in V to a constant. The constant being assigned to a variable v in V can be accessed via $A(v)$.*

Definition 2 *Let $(v_1,\ldots,v_n)$ be the parameter list of a function definition. Let V be a set of k ($\leq n$) variables taken from the parameter list. Let A_V be an assignment of variables in V. Then, we call an assignment A' to all parameters of the function a* complete extension *of the assignment A_V if for all $v \in V$, $A'(v) = A_V(v)$.*

We next define the term "function application" given our understanding of an assignment.

Definition 3 *Given a function definition fd with an assignment A to all parameters, an* application *of fd to the assignment A is defined as the invocation of the function fd with its arguments taking values from the assignment A.*

The correctness of our dependence analysis can thus be expressed as follows:

Theorem 1 (Dependence Analysis Safety) *Given a function library* $(fd_1, \ldots, fd_n)$. *Let* $\mathbf{F_{tab}} = \mathcal{BTA}_{\mathbf{P}}$ $(\mathbf{fd_1}, \ldots, fd_n)$. *For any function* fd_i *with name* f_i *and parameters* $(v_1, \ldots, v_n)$, *if* D_i *is the set of parameters involved in* $\mathbf{F_{tab}}(\mathbf{f_i}).\mathbf{ret}$, *then for any assignment to* D_i, A_{D_i}, *the following holds:*

> *For any two distinct and complete extension of* A_{D_i}, *denoted by* A_1 *and* A_2, *if application of* fd *to both the assignments* A_1 *and* A_2 *terminates, then the returned value in both applications are the same.*

A proof of the theorem can be constructed by a structural induction over the syntactic constructs of expressions. We omit the detail here.

4.3 Binding-time Constraints Generation and Simplification

We generate binding-time constraints for request/assert annotations based on the results from dependence analysis. The binding-time constraints are expressed in terms of the binding-time expressions of the variables stored in associated binding-time environment. More specifically,

- A request annotation `Request`e is converted to a binding-time constraint $bt_e = \mathtt{S}$ where bt_e is the binding-time expression of e.
- An assert annotation `Assert`ξ is converted to a binding-time constraint ξ

Thus the profitable specialization scenario of a function definition can be written as

$$\bigvee(\xi_{req_i}) \wedge \bigwedge(\xi_{ass_j})$$

where $\{\xi_{req_i}\}$ are the binding-time constraints generated for request annotations and $\{\xi_{req_i}\}$ are the binding-time constraints generated for assert annotations.

Figure 6 depicts an example of local binding-time environments and binding-time constraints for request/assert annotations over `mul` and `add` function definitions.

Source Code	Local BT Environments	BT Constraints
`int add(int m, int n){`		
`  return m + n;`	$[(\mathtt{m} \mapsto \mathtt{bt_m}), (\mathtt{n} \mapsto \mathtt{bt_n}), (\mathtt{ret} \mapsto \mathtt{bt_m} \sqcup \mathtt{bt_n})]$	
`}`		
`int mul(int x, int y){`		
`  int z;`	$[(\mathtt{z} \mapsto \mathtt{bt_z})]$	
`  if Request (x > 0){`	$[(\mathtt{x} \mapsto \mathtt{bt_x})]$	$\mathtt{btc_1} : \mathtt{bt_x} = \mathtt{S}$
`  z = mul(x − 1, y);`	$[(\mathtt{x} \mapsto \mathtt{bt_x}), (\mathtt{y} \mapsto \mathtt{bt_y}), (\mathtt{z} \mapsto \mathtt{bt_x} \sqcup \mathtt{bt_y})]$	
`  return add(z, y);`	$[(\mathtt{y} \mapsto \mathtt{bt_y}), (\mathtt{z} \mapsto \mathtt{bt_x} \sqcup \mathtt{bt_y},), (\mathtt{ret} \mapsto \mathtt{bt_x} \sqcup \mathtt{bt_y})]$	
`  }`		
`  if Request (x == 0)`	$[(\mathtt{x} \mapsto \mathtt{bt_x})]$	$\mathtt{btc_2} : \mathtt{bt_x} = \mathtt{S}$
`  return 0;`	$[(\mathtt{ret} \mapsto \mathtt{bt_x})]$	
`  if Request (y > 0){`	$[(\mathtt{y} \mapsto \mathtt{bt_y})]$	$\mathtt{btc_3} : \mathtt{bt_y} = \mathtt{S}$
`  z = mul(x, y − 1);`	$[(\mathtt{x} \mapsto \mathtt{bt_x}), (\mathtt{y} \mapsto \mathtt{bt_y}), (\mathtt{z} \mapsto \mathtt{bt_x} \sqcup \mathtt{bt_y})]$	
`  return add(z, x);`	$[(\mathtt{x} \mapsto \mathtt{bt_x}), (\mathtt{z} \mapsto \mathtt{bt_x} \sqcup \mathtt{bt_y}), (\mathtt{ret} \mapsto \mathtt{bt_x} \sqcup \mathtt{bt_y})]$	
`  }`		
`  return mul(−x, −y);`	$[(\mathtt{x} \mapsto \mathtt{bt_x}), (\mathtt{y} \mapsto \mathtt{bt_y}), (\mathtt{ret} \mapsto \mathtt{bt_x} \sqcup \mathtt{bt_y})]$	
`}`		

Global Table τ

`add` $\mapsto$ ([m, n], $\mathtt{bt_m} \sqcup \mathtt{bt_n}$)
`mul` $\mapsto$ ([x, y], $\mathtt{bt_x} \sqcup \mathtt{bt_y}$)

Figure 6: Example of Dependency Analysis and Binding-time Constraint Generation

The binding-time constraint representing the profitable specialization scenario for function `mul` is: $btc_1 \vee btc_2 \vee btc_3$. It is further simplified using distributivity and absorption laws. The final result is $(bt_x = \mathtt{S}) \vee (bt_y = \mathtt{S})$

5 Completing the Picture: Profitable Specialization

Once all relevant specialization scenarios are generated, specialization of functions will follow in a traditional manner. Since asserts at each function definition have been verified with respective to all scenarios, no additional check is needed during the actual specialization.

There are various ways to handle specialization of functions in the function library. One possi-

bility is to view the function library as a component, and package it as a *component generator*, as described in [2]. Once a concrete usage context has been established, the component generator facilitates the negotiation between different functions to elect the best specialization scenario at each function invocation, and performs the respective specialization using the elected scenario. Readers may wish to refer to [2] for detailed description of the technique.

6 Conclusion

In this paper, we describe a declarative mechanism that allow non-experts to declare inside a function definition a set of *requests* for certain specialization opportunity to be addressed. We also introduce the notation of *asserts* over binding-time constraint to rule out undesirable partial evaluation, such as infinite specialization. We then describe an algorithm to verify the consistency between these *requests* and *asserts*, generate a set of specialization scenarios aiming at achieving a safe and profitable specialization. We deem our approach to partial evaluation lightweight, because of its simplicity in allowing non-experts to declare their intention for specialization. In fact, by adopting a fixed set of requests (such as eliminating conditional tests) and asserts (such as poor man's generalization under dynamic control), we can fully automate the process of specialization scenario generation.

On the other hand, such a lightweight approach will not be able to perform sophisticated specialization fine-tuning, as provided by heavy-weight system such as that described in [14]. We believe that our system can be very effective in dealing with small-sized functions, such as those provided in function libraries and components. We are currently investigating the applicability of this approach in these domain.

References

[1] D. Bjorner, N. D. Jones, and A. P. Ershov. *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop, Gammel Avernaes, Denmark, 18-24 Oct., 1987*. Elsevier Science Inc., 1988.

[2] Gustavo Bobeff and Jacques Noy. Component specialization. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 39–50, New York, NY, USA, 2004. ACM Press.

[3] Anders Bondorf. Improving binding times without explicit cps-conversion. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 1–10, New York, NY, USA, 1992. ACM Press.

[4] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation: extended version. Technical Report Technical Report 93/4, University of Copenhagen, 1993.

[5] C. Consel. A tour of schism: a partial evaluation system for higher-order applicative languages. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154. ACM Press, 1993.

[6] Charles Consel and Siau Cheng Khoo. Parameterized partial evaluation. *ACM Transaction on Programming Languages and Systems*, 15(3):463–493, 1993.

[7] O. Danvy. Type-directed partial evaluation. In *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 242–257, 1996.

[8] M. Das. *Partial evaluation using dependence graphs*. PhD thesis, Computer Sciences Department, University of Wisconsin, 1998.

[9] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. The benefits and costs of dyc's run-time optimizations. *ACM Transaction on Programming Languages and Systems*, 22(5):932–972, 2000.

[10] Carsten Kehler Holst. Poor man's generalization, Aug 1998. Working Note, DIKU.

[11] N.D. Jones. Automatic program specialization: A re-examination from basic principles. In *In [1]*, pages 225–282, 1988.

[12] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993.

[13] N.D. Jones, P. Sestoft, and H. Sondergaard. An experiment in partial evaluation: the generation of a compiler generator. *Journal of LISP and Symbolic Computation*, 2(1):9–50, 1989.

[14] A.-F. Le Meur, J.L. Lawall, and C. Consel. Specialization scenarios: A pragmatic approach to declaring program specialization. *Higher-Order and Symbolic Computation*, 17(1):47–92, 2004.

[15] Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Specialization patterns. In *ASE '00: Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, page 197, Washington, DC, USA, 2000. IEEE Computer Society.