Towards Constructing Reusable Specialization Components

Ping Zhu Siau-Cheng Khoo

Department of Computer Science
National University of Singapore
{zhuping,khoosc}@comp.nus.edu.sg

Abstract

Component-based software development advocates the reuse of generic off-the-shelf components to build complex and reliable applications. Unfortunately, the genericness of components results in degradation of system performance. Little progress has been made in promoting the specialization of a component independent of its use context. In this paper we propose a component specialization framework aiming at producing *reusable specialization component* which are adaptive to different specialization contexts. We advocate *profitability declaration*, a novel methodology to capture specialization opportunities independent of how components are deployed. This conceptual profitability declaration is translated into a *profitability signature* in the form of the binding-time constraint. A *profitable specialization component*, PSC for short, is then developed, aiming to be deployed in various applications in place of the original generic component, as well as to be adaptive to different specialization contexts. In addition to the merit of reusability, PSC also achieves a reasonable balance between multiplicity of specialized codes and the space required for keeping them. We believe that our framework will promote the usage of program specialization in component-based software development.

Categories and Subject Descriptors D.1.2 [*Automatic Programming*]: Program synthesis; D.2.10 [*Design*]: Methodologies; D.2.m [*Miscellaneous*]: Reusable software

General Terms Languages, Design

Keywords Program specialization, component-based software development

1. Introduction

Component-based software development, CBSD for short, advocates the philosophy that software developers should reuse generic off-the-shelf components to build complex and reliable applications [16, 21]. It has been proven to be an effective paradigm in contemporary software industry. However, the genericness of components results in degradation of system performance, which has been recognized in many areas such as operating systems and graphics. One such performance inefficiency stems from the fact that a component may be repetitively executed with respect to multiple use contexts, each of which shares common partial configurations.

As an automatic program specialization technique that specializes a code with respect to its partial invariant input in earlier stage (*aka. compile-time*) and produces a more efficient *specialized component*, partial evaluation [17] has been used to tackle this common inefficiency in recent decades. Specifically, partial evaluation has usually been performed over an *application* into which generic components are reused. While this approach produces efficient specialized codes for applications, it requires generic components to be specialized multiple times with respect to different specialization contexts which are only established in multiple applications. We term this approach *application-driven component specialization*.

The technique of application-driven component specialization is in conflict with the spirit of software reuse: Components are not specialized once and then used in multiple applications. Instead, repetitive and thus redundant specialization of components is necessary for different applications.

Creating an application-independent specialization component, which is adaptive to different specialization contexts, on the other hand, has several technical challenges.

Firstly, code specialization – partial evaluation in particular – is typically driven by *initial specialization contexts* which become explicit only when a component is deployed in an application. Since a component implementation typically performs case analysis over its use contexts, it inhibits effective specialization in the absence of the initial specialization contexts. There have been several works that allow component developers to manually declare specific specialization contexts for components, such as [18] for imperative programs (rather than components) and [11] for objects and classes. In [5], Bobeff et al. proposed a component specialization approach which systematically creates an exhaustive list of specialization contexts for a component, then allows component developers to intervene by manually removing those inconsistent specialization contexts or unsuitable specialization contexts by considering benefits in terms of specialization opportunities. Unfortunately, this intervention makes the process of generating suitable specialization contexts an art only mastered by programmers with in-depth knowledge about partial evaluation.

Given that multiple specialization contexts are required for application-independent component specialization, the second challenge is to be able to manage specialized codes that are produced with respect to the multiple specialization contexts, even before deploying them in various applications. As these specialized codes are generated beforehand, they enable maximal reuse across multiple applications. On the other hand, since the specialized codes are generated with respect to many different contexts, they exist in multiplicity. It thus becomes important to manage and balance the trade-off between the multiplicity of specialized codes and the space required for keeping them. In this respect, Bobeff et al. developed a *component generator generator* which takes in a binding-time annotated component and creates a generating extension, called *service generator* for each available specializa-

tion context. These service generators are then grouped together to form a *component generator*, which is then used by applications. This technique inevitably produces significant amount of redundant codes among the set of service generators created. Other existing techniques in managing multiple specialized codes, such as [18, 3, 11, 25, 1, 24], have the same disadvantage.

In this research, we envisage a specialization framework in which specialization opportunities can be uncovered at the component level, independent of the applications in which components are reused. Furthermore, specialization, in addition to analysis, can be performed to a good extent at component level, decreasing the amount of repeated specialization of components at application level. Our vision is thus to *replace a component with a specialization component that caters for multiple specialization opportunities, while minimizing the need for repetitive and redundant component specialization at the application level.*

Despite our desire to conduct application-independent component specialization, we continue to value the existence of application in providing specific use contexts for components. Therefore, we require a seamless integration of both *component* and *application* specialization processes in our specialization framework.

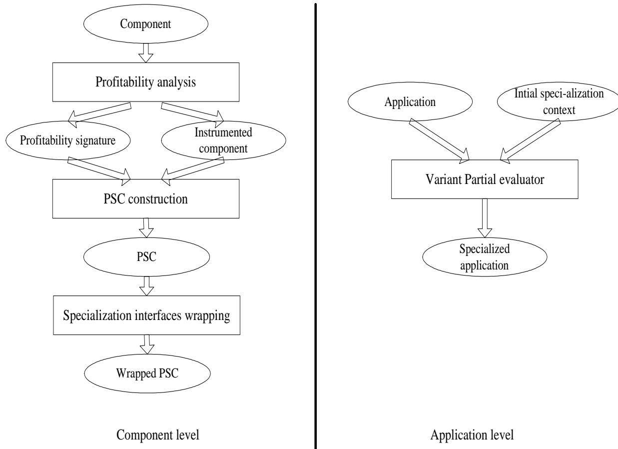

Figure 1. Profitability-oriented Component Specialization Framework

Figure 1 depicts our vision for component specialization. It shows two levels of specialization processes. At the component level we subject a component to a *profitability analysis*. The analysis result is then used for specialization through a *PSC construction*, yielding a *profitable specialization component*, PSC for short. The PSC is further wrapped with some common specialization interfaces to form a wrapped PSC. At the application level, instead of specializing the deployed generic components with respect to their respective use contexts propagated from the initial specialization context, the corresponding wrapped PSCs of the involved components are selected for specialization purpose, and the entire application is subject to a variant of application-driven partial evaluation to produce the final specialized application.

Central to our framework are two novel special processes: *Profitability analysis* and *PSC construction*. These two processes aim to address the two challenges mentioned earlier:

- Profitability analysis aims to discover the condition in which the *specialization opportunities* of a component can be fulfilled. A specialization opportunity identifies a component code segment the elimination of which during specialization can lead to the generation of more effective specialized code. In this paper, we identify the *codes for conditional tests* in a component with specialization opportunities. Profitability analysis outputs a collection of specialization contexts, called *binding-time signatures*, which enables the elimination of the corresponding conditional tests.
- PSC construction aims to manage multiple specialized codes, resulting from specializing a component with respect to multiple binding-time signatures. The central idea of such management is *code sharing*. At the component level, a specialized code is represented by annotating the original code with an action analysis result. PSC construction partitions each action-annotated code into pieces and identifies *sharable codes* which are identical specialized codes with respect to different binding-time signatures. PSC shares these pieces across different specialized code instances, thus capturing multiple specialized codes through *webs of annotated code segments*. PSC construction thus achieves time and space savings in managing multiple specialized codes. This advantage distinguishes our framework from existing frameworks in synthesizing multiple binding-time signatures with original component [18, 5, 3, 11, 25, 1, 24].

  The results of PSC construction are called pre-GEs, to signify that they are not full generating extensions, and further specialization is required (at application level) to obtain the desired generating extension.

The rest of this paper is outlined as follows: Section 2 introduces the core language and some notational conventions. In Section 3, we explain how profitability declarations guide component specialization through some representative examples and then present the profitability analysis to convert the conceptual profitability declaration to profitability signature. Next we elaborate our proposal in constructing and deploying PSC in applications in Section 4. Before concluding, we survey related works in the area of component specialization in Section 5.

2. Preliminaries

In this paper we choose the component model as a function definition which may be interrelated with other function definitions. The terms *component* and *function definition* are used interchangeably in the following sections. The core language is a subset of C language and its abstract syntax is defined in Figure 2.

v	$\in$	**Var**	Variables
f, g	$\in$	**FName**	Function names
n	$\in$	$\mathcal{N}$	Numerals
b_{op}	$\in$	**BOp**	Binary operators
	$::=$	$+ \mid - \mid * \mid / \mid ==$ $\mid != \mid < \mid > \mid >=$ $\mid <= \mid \&\& \mid \|\|$	
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
	$::=$	$s_1; s_2 \mid \texttt{while}\ e\ s \mid \texttt{return}\ e$ $\mid v = e \mid \texttt{if}\ e\ s_1\ \texttt{else}\ s_2$	
$decl$	$\in$	**Decl**	Declarations
	$::=$	$\texttt{int}\ v$	
fd	$\in$	**FDef**	Function definitions
	$::=$	$\texttt{int}\ f\ (decl^+)\ \{decl^*\ ;\ s\}$	

Figure 2. Syntax of the Core Language - A C subset

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

definition must return a value. The essential features of our framework, as described earlier, can be thoroughly investigated through this core language.

The notational convention for expressing binding-time information is listed as follows. 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
	$::=$	$\mathtt{S} \mid \mathtt{D} \mid bt_v \mid bt_{e_1} \sqcup bt_{e_2} \mid bt_{e_1} \sqcap bt_{e_2}$	

There are three basic binding-time expressions: two binding-time constants S and D representing the static and the dynamic values respectively, and a binding-time variable bt_v ranging over S and D. These are ordered in decreasing staticness: $\mathtt{S} \sqsubset bt_v \sqsubset \mathtt{D}$. A composite binding-time expression is formed using two operators: least upper bound $\sqcup$ and greatest lower bound $\sqcap$. The ordering can be naturally extended to partial ordering over tuples of binding-time expressions.

We do **not** specialize a component with respect to concrete values, as such values are only provided at the application level. Instead, binding-time information about component parameters is used in component specialization. We highlight two terminologies which represent two categories of binding-time information under two different circumstances:

- *Binding-time signature*: It is a contract-like binding-time information associated with original component independent of component use context. This information is derived at component level.
- *Binding-time context*: It is a context information describing the binding-time information established at component deployment time.

3. Profitability Analysis

As pointed out in Section 1, the current application-driven approach is inimical to producing effective and yet reusable specialization components. In our effort to make component specialization application-independent, we inevitably shift the focus of specialization from the traditional objective of "how best to prepare a piece of code for specialization in a specific context propagated from an initial specialization context" to the new objective of *"how best to prepare a piece of code for specialization such that the specialized code remains effective in as many applications as possible."* This new perspective enables component developers to take advantage of the knowledge of the component implementation and prepare suitable specialization contexts for future deployment.

There can be various kinds of application-independent specialization opportunities for components, not all need to be handled by partial evaluation technique. For instance, an application developer may wish to specialize a component handling generic array operation when the latter is deployed in a context that deals with bit-vectors.

In this paper, we investigate the technique of turning specialization profitability of a component into a sufficient context that is amenable to partial evaluation. Specifically, we advocate using the term "profitability" to indicate the opportunity for *specialization*, i.e. *the ability to specialize conditional tests away*. This is based on a reasonable heuristic that static reduction of conditions and static unrolling of loops are primary sources of profitable specialization both in terms of time and space. This profitability can be further divided into two categories:

- *Direct profitability*: The ability to directly specialize a conditional test inside a function definition away.
- *Indirect profitability*: The ability to specialize a function call so that the (direct or indirect) profitability inside the called function definition may be reaped.

Profitability fulfillment stands for the fact that the binding-time expression of conditional test is S or the binding-time context established at the function call site is deemed *profitable* with respect to a binding-time signature of the corresponding function definition.

As a function's return value can be used to determine the truth value of a conditional test, it can be involved in determining the profitability of a conditional test – as a nested function call within the test. Consequently, the binding-time information of a function's return value can also be a source of profitability. For such a case, we require that the binding-time context established at the conditional test should not only enable the nested function call be specialized profitably but also enable the function call return static value so that so that the whole conditional test could produce static value.

Profitabilities residing inside a component can be identified by *profitability points*. A profitability point refers to a program segment which possesses direct or indirect profitability. According to the classification of profitability as described above, there are also two categories of profitability points: Conditional tests (which may contain nested function calls) and function call expressions.

Consider the following two interrelated function definitions.

```
int f(int x, int y){
  if x > 0               /* profitability point 1*/
  return y+1;
  else return y-1;
}

int g(int x, int y){
  if y > 0               /* profitability point 2*/
  return f(x,y);         /* profitability point 3*/
  else return 0;
}
```

In this example, function f has one profitability point and function g has two profitability points, as highlighted in the in-lined comments. We name a function definition without any profitability point as a *plain function*.

Correspondingly, a specialization that fulfills all or part of the (direct and/or indirect) profitabilities available in a component is called a *profitable specialization*. Otherwise, it is termed as *unprofitable specialization*.

3.1 Profitability Signature

We convert a conceptual profitability declaration into a piece of binding-time information expressed in the form of binding-time constraint. This binding-time information stipulates a binding-time condition required for function parameters in order to achieve profitability fulfilment within a function definition. The syntax of binding-time constraint is as follows:

ξ	$\in$	$\mathbf{BT_c}$	BT constraints
	$::=$	$\mathtt{TRUE} \mid \mathtt{FALSE} \mid bt_{e_1} = bt_{e_2}$	
		$\mid \xi_1 \wedge \xi_2 \mid \xi_1 \vee \xi_2$	

We denote the set of binding-time variables occurring in a binding-time constraint ξ by $V(\xi)$. This is understood to be a subset of the set of binding-time variables $\mathbf{Var}_{BT}$ (such as those pertaining to function parameters) which is of interest in formulating the constraints. Semantically, a *satisfiable binding-time valuation* of ξ is an assignment of either S or D to all binding-time variables in $\mathbf{Var}_{BT}$ such that the constraint becomes true. Such an assignment can be expressed as a conjunction of equality constraints between binding-time variables and binding-time constants.

Given a set of binding-time variables $\mathbf{Var}_{BT}$, the constraint `FALSE` signifies that there is no satisfiable binding-time valuation, and the constraint `TRUE` signifies that all possible binding-time valuations are satisfiable.

Given two valuations of ξ, ϑ_1 and ϑ_2, we say that $\vartheta_1 \sqsubseteq \vartheta_2$ iff for all $bt_v \in \mathbf{Var}_{BT}$, $\vartheta_1(bt_v) \sqsubseteq \vartheta_2(bt_v)$.

The set of binding-time signatures capturing the profitability declaration of a function is termed *profitability signature*. The profitability signatures derived for functions `f` and `g` defined in the beginning of Section 3 are:

$$\begin{array}{ll} \xi_f : & bt_x = \mathtt{S} \\ \xi_g : & (bt_x = \mathtt{S}) \vee (bt_y = \mathtt{S}) \end{array}$$

ξ_f states that as long as the binding-time value of the parameter `x` is `S`, the profitability at point 1 can be fulfilled, regardless of the binding-time value of the parameter `y`. This signature can also be expressed equivalently using all binding-time variables associated with `f`'s parameters, as follows:

$$\xi_f : \quad (bt_x = \mathtt{S} \wedge bt_y = \mathtt{S}) \vee (bt_x = \mathtt{S} \wedge bt_y = \mathtt{D})$$

ξ_g expresses a disjunctive condition in which the profitability opportunities at point 2 and point 3 can be fulfilled respectively.

The profitability signature of a plain function is encoded as `FALSE`. In other words, there is no binding-time valuation of the parameters that can make specialization of a plain function profitable.

Implementation-wise, profitability signatures are not generated by a typical forward-fashion binding-time analysis; instead, they are generated by propagating outwardly those "binding-time requests" at the profitability points. The algorithm for generating profitability signature is presented in Section 3.3.

3.2 Specialization Policy

Before detailing the algorithm for generating profitability signatures, we explain the *specialization policy* we have adopted in using such signatures for specialization. The rationale for discussing the policy here is that it will provide the direction in which we design and discover profitability signatures.

Specialization is driven by the binding-time *contexts* available at program points. A binding-time context is a simple binding-time constraint represented as $\bigwedge\{bt_{v_i} = \mathtt{S} \mid \mathtt{D}\}$, where $\{bt_{v_i}\}$ is the set of binding-time variables pertaining to the function parameters. It can be related to profitability signature through an *entailment* relation defined over binding-time constraints, as follows:

DEFINITION 1 (Entailment Relation). *Let $\mathbf{Var}_{BT}$ be the set of binding-time variables in the domain of discourse. Let $\mathcal{V}(\xi)$ be the set of satisfiable binding-time valuations with respect to a binding-time constraint ξ. A binding-time constraint ξ_1 is said to* be entailed by *another binding-time constraint ξ_2, denoted by $\xi_1 \vdash \xi_2$, iff for any valuation $\vartheta_1 \in \mathcal{V}(\xi_1)$, there exists a valuation $\vartheta_2 \in \mathcal{V}(\xi_2)$ such that $\vartheta_1 \sqsubseteq \vartheta_2$.*

As an example, a binding-time constraint $\xi_1 = (bt_x = \mathtt{S} \wedge bt_y = \mathtt{S})$ is entailed by another constraint $\xi_2 = (bt_x = \mathtt{S} \wedge bt_y = \mathtt{D})$.

Since a binding-time valuation is an assignment of binding-time values to binding-time variables, it can be treated as a binding-time (equality) constraint, and be involved in establishing entailment relations with other constraints. This is used in the next definition:

DEFINITION 2 (Minimal Profitable Context). *Let $\mathbf{Var}_{BT}$ be the set of binding-time variables in the domain of discourse, ϑ be a binding-time valuation, and ξ be a binding-time constraint. Given that $\vartheta \vdash \xi$, a* minimal profitable context *of ϑ in ξ is a valuation ϑ_m such that:*

$$\begin{array}{l} \vartheta \sqsubseteq \vartheta_m \wedge \vartheta_m \in \mathcal{V}(\xi) \wedge \\ \quad \forall \vartheta' \in \mathcal{V}(\xi) : (\vartheta_m \sqsubseteq \vartheta' \vee \ \neg(\vartheta_m \sqsubseteq \vartheta' \vee \vartheta' \sqsubseteq \vartheta_m)) \end{array}$$

As another example, for $\xi_3 = (bt_x = \mathtt{S} \wedge bt_y = \mathtt{S})$ and $\xi_4 = ((bt_x = \mathtt{S} \wedge bt_y = \mathtt{D}) \vee (bt_x = \mathtt{D} \wedge bt_y = \mathtt{S}))$, we have $\xi_3 \vdash \xi_4$. Furthermore, the minimal profitable context of ξ_3 in ξ_4 can either be $\xi_m = (bt_x = \mathtt{S} \wedge bt_y = \mathtt{D})$ or $\xi_m = (bt_x = \mathtt{D} \wedge bt_y = \mathtt{S})$.

Now, the specification policy can be stated as follows:

> *Given a function f and its associated profitable signature ξ_P. If a binding-time context ξ_1 for a call to f is entailed by ξ_P, then the function call will be specialized with respect to a minimal profitable context of ξ_1. Otherwise, all the binding-time values of f-call's arguments, as well as that of f-call's returned value, will be classified as* `D`.

We demonstrate how the specialization policy governs the component specialization process by considering the following simple examples:

Example 1

Supposed the function `f` defined earlier is called at two locations with different binding-time contexts $c_1 : (bt_x = \mathtt{S} \wedge bt_y = \mathtt{D})$ and $c_2 : (bt_x = \mathtt{S} \wedge bt_y = \mathtt{S})$ respectively, where bt_x and bt_y are binding-time variables pertaining to the binding-time information of the first argument and second argument of function call `f`. c_1 and c_2 are both minimal profitable context of ξ_f. Thus each of the two calls will be specialized with respect to the exact binding-time context correspondingly.

There is no intention to collapse the specializations of these two calls into one specialized code, even though the two contexts only differ in their binding-time values for parameter `y`. This policy allows us to explore static information as much as possible in the presence of profitable specialization.

Example 2

Consider the following code:

```
int add(int m, int n){
  return m+n;
}

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

Function `add` is a plain function and its profitability signature is encoded as `FALSE`. The two calls to `add` in the function definition of `mul` can only be specialized with all their arguments as dynamic regardless of the binding-time values that `x`, `y` and `z` hold before the calls. After the call, the binding-time values of the arguments of `add` will be restored to those values before the call. This treatment of temporarily assigning the binding-time value of an argument to

D is similar to the "raise" operation used in many existing partial evaluators [6].

3.3 Profitability-oriented BTA

We have developed and implemented a modular profitability-oriented BTA to compute profitability signatures of components (Figures 3, 4 and 5). In line with the treatment found in conventional BTA, this analysis is both flow-sensitive and context-sensitive. Moreover, the specialization policy also requires this novel BTA to take into consideration the *raise condition* that: a function call should be specialized with respect to either the minimal profitable context of a binding-time context or the configuration that all its argument should be D. This raise condition may affect the binding-time value of a function call's return value, which is also a candidate for profitability point as we have explained earlier.

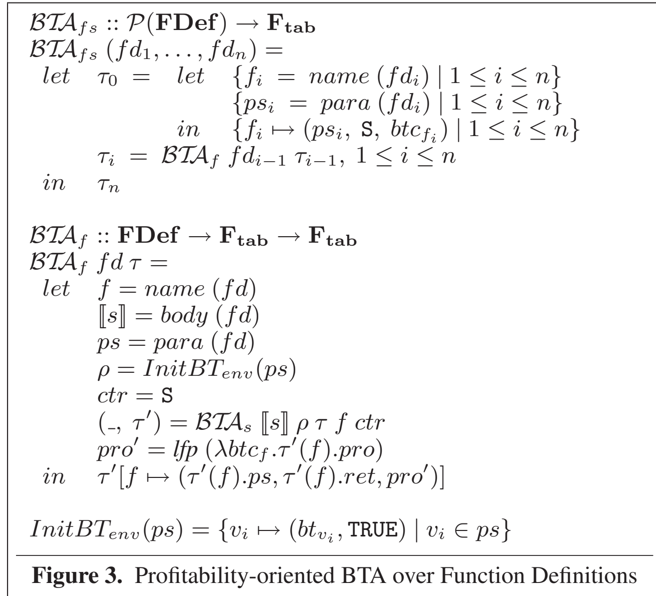

Figure 3. Profitability-oriented BTA over Function Definitions

The profitability-oriented BTA operates over a set of interrelated function definitions, and returns:

- At each program point, a local binding-time environment ρ, of the form $[v \mapsto (bte,\ btc_e)]$. v ranges over program variables occurring at the program point; *bte* is a binding-time expression of v, computed by taking into consideration both the data- and control-flow dependencies. btc_e is a binding-time constraint determining when to perform a "raise" operation on variable v during the analysis; specifically, if btc_e cannot be satisfied by the context in which v is used, the *bte* is raised to D to perform the "raise" operation.

  The set of local binding-time environments collected at program points instruments the original component to facilitate construction of PSC.

- A global function name indexed table $\tau \in \mathbf{F_{tab}}$, of the form $[f \mapsto (ps,\ ret,\ pro)]$. 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; pro is a disjunction of binding-time constraint standing for the function's profitability signature.

The key task for profitability-oriented BTA is to generate binding-time constraint at each of the profitability points to establish a condition of *profitability fulfillment* as we explained in Section 3. $\mathcal{BTA}_s$ (Figure 4) and $\mathcal{BT}_e$ (Figure 5) are responsible for generating binding-time information for statements and expressions respectively. Specifically, they play central roles in generating binding-time constraints for conditional tests and function calls respectively. The disjunction of these binding-time constraints is stored into the pro field of the corresponding entry in τ. As function definitions may be recursive, the ultimate profitability signature is obtained by performing the least fix-point operation (*lfp*) over pro field.

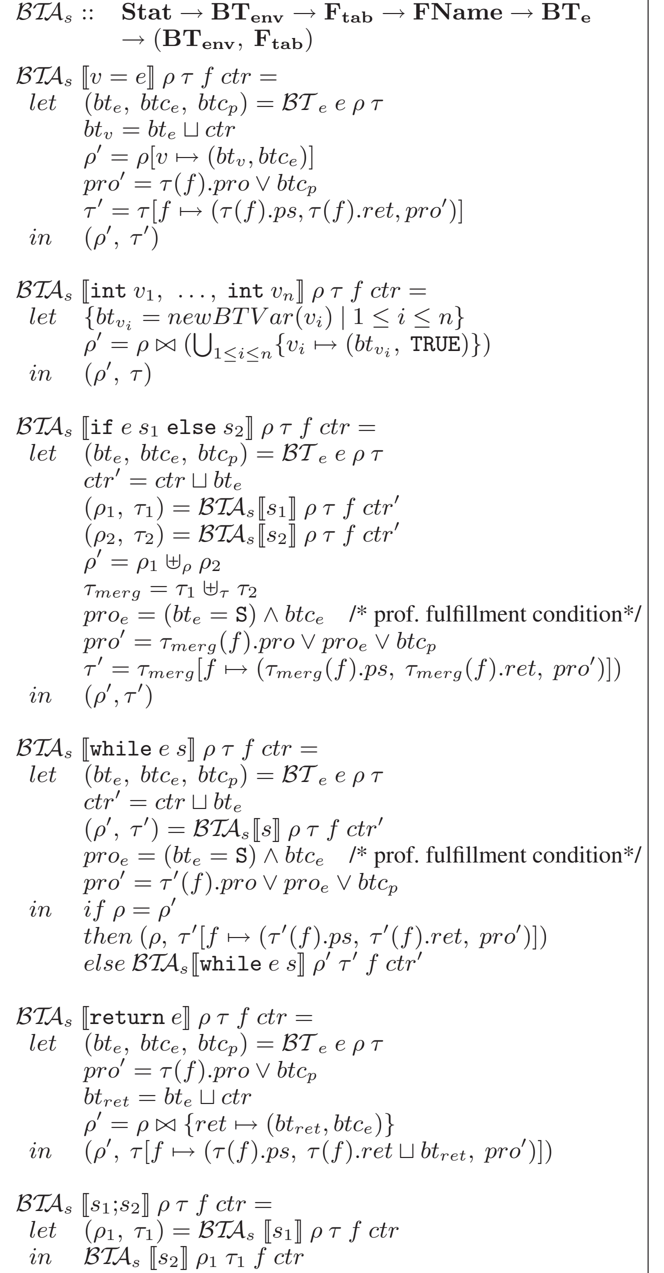

Figure 4. Profitability-oriented BTA over Statement

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

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

$$\mathcal{BT}_e :: \mathbf{Exp} \rightarrow \mathbf{BT_{env}} \rightarrow \mathbf{F_{tab}} \rightarrow (\mathbf{BT_e},\ \mathbf{BT_c},\ \mathbf{BT_c})$$

$$\mathcal{BT}_e\ [\![n]\!]\ \rho\ \tau = (\mathtt{S},\ \mathtt{TRUE},\ \mathtt{FALSE})$$

$$\mathcal{BT}_e\ [\![v]\!]\ \rho\ \tau = (\rho(v).bte,\ \rho(v).btc_e,\ \mathtt{FALSE})$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{BT}_e\ [\![e_1\ b_{op}\ e_2)]\!]\ \rho\ \tau =} \\ let & (bt_{e_1},\ btc_{e_1},\ btc_{p_1}) = \mathcal{BT}_e[\![e_1]\!]\ \ \rho\ \tau \\ & (bt_{e_2},\ btc_{e_2},\ btc_{p_2}) = \mathcal{BT}_e[\![e_2]\!]\ \ \rho\ \tau \\ in & (bt_{e_1} \sqcup bt_{e_2},\ btc_{e_1} \wedge btc_{e_2},\ btc_{p_1} \vee btc_{p_2}) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{BT}_e\ [\![f\ (e_1, \ldots, e_n)]\!]\ \rho\ \tau =} \\ let & \{(bt_{e_i},\ btc_{e_i}\ btc_{p_i}) = \mathcal{BT}_e[\![e_i]\!]\ \ \rho\ \tau \mid 1 \leq i \leq n\} \\ & (v_1, \ldots, v_n) = \tau(f).args \\ & bte = \tau(f).ret[v_1 \mapsto bt_{e_1}, \ldots, v_n \mapsto bt_{e_n}] \\ & btc_f = \tau(f).pro \wedge (v_1 = bt_{e_1}) \ldots \wedge (v_n = bt_{e_n}) \\ & \quad \text{/* prof. fulfillment condition*/} \\ & btc_e = btc_f \wedge btc_{e_1} \wedge \ldots \wedge btc_{e_n} \\ & btc_p = btc_f \vee btc_{p_1} \vee \ldots \vee btc_{p_n} \\ in & (bte,\ btc_e,\ btc_p) \end{array}$$

Figure 5. Computing BT Expressions

- $\rho_1 \uplus_\rho \rho_2 = \{x \mapsto (bte_1 \sqcup bte_2, btc_{e_1} \wedge btc_{e_2}) | x \mapsto (bte_1, btc_{e_1}) \in \rho_1, x \mapsto (bte_2, btc_{e_2}) \in \rho_2\}$.
- $\tau_1 \uplus_\tau \tau_2 = \{f \mapsto (ps, ret_1 \sqcup ret_2, pro_1 \vee pro_2) | f \mapsto (ps, ret_1, pro_1) \in \tau_1, f \mapsto (ps, ret_2, pro_2) \in \tau_2\}$.

The soundness of the analysis can be expressed in terms of *profitable specialization*:

DEFINITION 3 (Profitable Specialization). *Specialization of f with respect to a context ξ is said to be a* profitable specialization *if either one of the following conditions hold during specialization:*

1. *One of f's direct profitability points will have the binding-time value of its test evaluated to* S*, or*
2. *One of the function calls within the body of f will be specialized with respect to a context ξ' that will result in profitable specialization.*

THEOREM 1 (Soundness). *Given a program $P = \{fd_1, \ldots, fd_n\}$, let $\mathcal{BTA}_{fs}(P) = \tau$. For any i between 1 and n, let f_i be the name of fd_i with parameter list $(x_1, \ldots, x_m)$. For any call context ξ_i of f_i such that $\xi_i \vdash \tau(f_i).pro$, specializing f_i with respect to ξ_i will be profitable.*

Figure 6 depicts an example of local binding-time environments and binding-time constraints for profitability obtained from profitability-oriented BTA over the `mult` and `add` components.

Binding-time constraints btc_3 and btc_7 can be simplified to `FALSE` based on the knowledge that $\tau(\mathtt{add}).pro = \mathtt{FALSE}$. The other binding-time constraints can be simplified using the distributive and the absorption laws. The simplified result for the profitability signature of `mul` is thus:

$$\tau(\mathtt{mul}).pro = ((bt_x = \mathtt{S}) \vee \tau(\mathtt{mul}).pro \vee (bt_y = \mathtt{S}))$$

Finally, we conduct least fixed-point operation on $\tau(\mathtt{mul}).pro$ to obtain the ultimate profitability signature:

$$\tau(\mathtt{mul}).pro = ((bt_x = \mathtt{S}) \vee (bt_y = \mathtt{S}))$$

3.4 Discussion

In line with the idea of "design by contract" [19], the profitability signature can act as a contract provided by a component. This contract stipulates the condition under which effective specialization can take place. For simplicity, we do not consider the termination aspect of partial evaluation, which is usually controlled by adjusting the binding-time signatures of functions. Certainly, existing techniques in ensuring termination of partial evaluation (such as [2, 14]) can be added to our analysis. Alternatively, annotations can be included in the component body to ensure that the binding-time signature generated does not lead to non-termination of partial evaluation process. One such technique is to introduce an *assert* annotation which asserts that the binding-time values of some (usually non-inductive) parameters of a function be more dynamic than other (inductive) parameters. Interested readers may wish to refer to [26] for details.

Our specialization policy, which only allows specialization when the specialization context is entailed by the profitability signature, may be perceived as a form of under-specialization. There are certainly cases when a specialization context is not in harmony with profitability signature and yet the specialization can be considered effective. In fact, one may claim that profitability is a subjective concept that is determined by the programmer's specialization intention [12, 13]. We believe the current definition of profitability, which guarantees elimination of tests in if- and while-constructs, is acceptable as being practical by the partial evaluation community.

4. Profitable Specialization Component

In this section, we elaborate our method for synthesizing a reusable specialization component from the results of profitability-oriented BTA. We term this synthesized component as *profitable specialization component*, PSC for short. The objectives of PSC construction are as follows:

1. The original component should be transformed to a form that is both as close to the ultimate specialized code as possible and as reusable as possible. This is done only with the knowledge of profitability signature. It is noteworthy to point out that at component level, there is no information of concrete values passing to the component, neither is there any information about the *initial* binding-time context for component specialization.
2. PSC should occupy as minimal code space as possible, given that there can be multiple specialized codes available in the specialization component to meet the requirement of deploying the specialized component in different applications wrt. different specialization contexts.
3. PSC should be easily deployed in different applications, contributing to minimal administrative cost during partial evaluation of application application level.

In our framework, a component therefore exists in three forms: Its original generic component form, its corresponding PSC which aims to support multiple profitable specialization in future, and its specialized form, which is instantiated from PSC with respect to a specific specialization context propagated from the initial specialization context pertaining to an application.

4.1 Action-Annotated Components

The basic structure of PSC is an action-annotated code. We tread on the path taken by several popular partial evaluators such as Schism [7, 8] and Tempo [9] by employing action analysis to aid specialization. Given a piece of code, action analysis annotates each program construct with an action value, which is a clear instruction dictating the construction of specialized code. The action domain $\mathbf{AC_{val}}$ comprises four values `ev`, `rd`, `rc`, and `id`, which represent the actions *evaluation, reduce, reconstruct* and *identity* respectively. The action value of each program construct is strictly determined by its

Source Code	Local Environments	BT Constraints for Profitability
`int add(int m, int n){`		
`  return m+n; }`	$[(\mathtt{m} \mapsto (\mathtt{bt_m}, \mathtt{TRUE})), (\mathtt{n} \mapsto (\mathtt{bt_n}, \mathtt{TRUE})), (\mathtt{ret} \mapsto (\mathtt{bt_m} \sqcup \mathtt{bt_n}, \mathtt{TRUE}))]$	
`int mul(int x, int y){`		
`  int z;`	$[(\mathtt{z} \mapsto (\mathtt{bt_z}, \mathtt{TRUE}))]$	
`  if (x > 0){`	$[(\mathtt{x} \mapsto (\mathtt{bt_x}, \mathtt{TRUE}))]$	$\mathtt{btc_1} : \mathtt{bt_x} = \mathtt{S}$
`    z = mul(x-1,y);`	$[(\mathtt{x} \mapsto (\mathtt{bt_x}, \mathtt{TRUE})), (\mathtt{y} \mapsto (\mathtt{bt_y}, \mathtt{TRUE})), (\mathtt{z} \mapsto (\mathtt{bt_x} \sqcup \mathtt{bt_y}, \xi_1))]$	$\mathtt{btc_2} : \tau(\mathtt{mul}).\mathtt{pro} \wedge (\mathtt{bt_x} = \mathtt{bt_x}) \wedge (\mathtt{bt_y} = \mathtt{bt_y})$
`    return add(z, y);}`	$[(\mathtt{y} \mapsto (\mathtt{bt_y}, \mathtt{TRUE})), (\mathtt{z} \mapsto (\mathtt{bt_x} \sqcup \mathtt{bt_y}, \xi_1)), (\mathtt{ret} \mapsto (\mathtt{bt_x} \sqcup \mathtt{bt_y}, \xi_1 \wedge \xi_2))]$	$\mathtt{btc_3} : \tau(\mathtt{add}).\mathtt{pro} \wedge (\mathtt{bt_x} \sqcup \mathtt{bt_y} = \mathtt{bt_m}) \wedge (\mathtt{bt_y} = \mathtt{bt_n}) \wedge \xi_1$
`  if (x == 0)`	$[(\mathtt{x} \mapsto (\mathtt{bt_x}, \mathtt{TRUE}))]$	$\mathtt{btc_4} : \mathtt{bt_x} = \mathtt{S}$
`    return 0;`	$[(\mathtt{ret} \mapsto (\mathtt{bt_x}, \mathtt{TRUE}))]$	
`  if (y > 0){`	$[(\mathtt{y} \mapsto (\mathtt{bt_y}, \mathtt{TRUE}))]$	$\mathtt{btc_5} : \mathtt{bt_y} = \mathtt{S}$
`    z = mul(x,y-1);`	$[(\mathtt{x} \mapsto (\mathtt{bt_x}, \mathtt{TRUE})), (\mathtt{y} \mapsto (\mathtt{bt_y}, \mathtt{TRUE})), (\mathtt{z} \mapsto (\mathtt{bt_x} \sqcup \mathtt{bt_y}, \xi_1))]$	$\mathtt{btc_6} : \tau(\mathtt{mul}).\mathtt{pro} \wedge \mathtt{bt_x} = \mathtt{bt_x} \wedge \mathtt{bt_y} = \mathtt{bt_y}$
`    return add(z,x);}`	$[(\mathtt{x} \mapsto (\mathtt{bt_x}, \mathtt{TRUE})), (\mathtt{z} \mapsto (\mathtt{bt_x} \sqcup \mathtt{bt_y}, \xi_1)), (\mathtt{ret} \mapsto (\mathtt{bt_x} \sqcup \mathtt{bt_y}, \xi_1 \wedge \xi_2))]$	$\mathtt{btc_7} : \tau(\mathtt{add}).\mathtt{pro} \wedge (\mathtt{bt_x} \sqcup \mathtt{bt_y} = \mathtt{bt_m}) \wedge (\mathtt{bt_y} = \mathtt{bt_n}) \wedge \xi_1$
`  return mul(-x, -y);}`	$[(\mathtt{x} \mapsto (\mathtt{bt_x}, \mathtt{TRUE})), (\mathtt{y} \mapsto (\mathtt{bt_y}, \mathtt{TRUE})), (\mathtt{ret} \mapsto (\mathtt{bt_x} \sqcup \mathtt{bt_y}, \xi_1))]$	$\mathtt{btc_8} : \tau(\mathtt{mul}).\mathtt{pro} \wedge \mathtt{bt_x} = \mathtt{bt_x} \wedge \mathtt{bt_y} = \mathtt{bt_y}$

where $\xi_1 = \tau(\mathtt{mul}).\mathtt{pro} \wedge (\mathtt{bt_x} = \mathtt{bt_x}) \wedge (\mathtt{bt_y} = \mathtt{bt_y})$;
$\xi_2 = \tau(\mathtt{add}).\mathtt{pro} \wedge (\mathtt{bt_m} = \mathtt{bt_x} \sqcup \mathtt{bt_y}) \wedge (\mathtt{bt_n} = \mathtt{bt_y})$;
$\tau = [\mathtt{add} \mapsto ([\mathtt{m}, \mathtt{n}],\ \mathtt{bt_m} \sqcup \mathtt{bt_n},\ \mathtt{FALSE}),\quad \mathtt{mul} \mapsto ([\mathtt{x}, \mathtt{y}],\ \mathtt{bt_x} \sqcup \mathtt{bt_y},\ \mathtt{btc_1} \vee \ldots \mathtt{btc_8})]$

Figure 6. Profitability-oriented BTA result w.r.t `add` and `mul` Component

binding-time value. Action analysis enables decisions on how to specialize a code to be made at component analysis time rather than application specialization time. Moreover, action-annotated program is the desired product expected from run-time specialization, when no concrete values are available for conducting further specialization [9]. By specializing original component to one that is action-annotated, we have performed as much specialization as possible, and leaving the direct generation of specialized codes to be done at application level.

As mentioned in Section 3.1, given a function definition, profitability-oriented BTA outputs at every program point a local binding-time environment recording the binding-time expressions of program variables at that point, all in terms of the binding-time expressions of function parameters. This information is then further utilized to compute the action value of each syntactic construct. For the ease of presentation, we associate each syntactic construct with an *action-variable* which is an identifier referring to an *action-expression* whose value is computed by two auxiliary functions $\mathcal{A}_e$ and $\mathcal{A}_s$, which compute the action values for expression and statement respectively. The type signatures of these two functions are defined as follows:

$$\mathcal{A}_e :: \mathbf{Exp} \rightarrow \mathbf{BT_{exp}} \rightarrow \mathbf{AC_{val}}$$
$$\mathcal{A}_s :: \mathbf{Stat} \rightarrow \mathcal{P}(\mathbf{AC_{val}}) \rightarrow \mathbf{AC_{val}}$$

Interested readers may wish to refer to [8] for the detailed implementation of these two functions.

Figure 7 depicts an *action-variable* annotated code for functions `add` and `mul`.

4.2 PSC with Code Sharing

Action-variable annotated code is not the final form of PSC. We have yet to take advantage of the available profitability signature associated with the code. Recall that a profitability signature is a disjunction of binding-time constraints. Since these binding-time constraints are defined over parameters, and values of action variables are strictly determined by binding-time values, we can use each *ground* instance of a binding-time constraint to produce a (*variable-free*) *action-annotated code* from the original action-variable annotated code.

However, this naive approach may result in generating numerous action-annotated codes for a component with respect to a profitability signature. To minimize space usage, we adopt a novel approach to creating *sharable* action-annotated codes.

Our approach partitions an original action-variable annotated code into smaller units, say at the statement level, and allowing sharing of action-annotated *statements* among different specialized codes.

More specifically, we detect sharable action-annotated codes by looking up each statement of the component and its associated action-variable (i.e. action-expression) sequence. If two different binding-time signatures lead to an identical action value sequence, we term them as equivalent. Formally, we define an equivalence relationship between two binding-time signatures:

For a statement S and its associated action-expression sequence $\overline{A_{exp}}$, two binding-time signatures σ_1 and σ_2 are considered equivalent, denoted by $\sigma_1 \sim \sigma_1$, iff

$$\mathbf{Inst}(\overline{A_{exp}}, \sigma_1) = \mathbf{Inst}(\overline{A_{exp}}, \sigma_2).$$

The *instantiating* function **Inst** takes an action-expression sequence $\overline{A_{exp}}$ and a specific binding-time signature σ; it returns a action value sequence computed by applying the functions $\mathcal{A}_e$ and $\mathcal{A}_s$ to the action-expression elements-wise. **Inst** is defined as

$$\mathbf{Inst}(\overline{A_{exp}}, \sigma) = \{A_{exp}[\sigma] \mid A_{exp} \in \overline{A_{exp}}\}$$

We have developed an algorithm **FindEqSet** (presented in Figure 8) which takes in:

- $\overline{\overline{A_{exp}}}$: A sequence of action-expression sequences and
- $\overline{\sigma}$: A sequence of binding-time signatures.

and returns, for each statement in the code, a set of pairs. Each pair contains the following information:

```
      int add(int m^{A0}, int n^{A1}) {
1 :     return (m^{A2} + n^{A3})^{A4};^{A5}
      }

      int mul(int x^{M0}, int y^{M1}) {
1 :     int z^{M2};^{M3}
2 :     if (x^{M4} > 0^{M5})^{M6} {
3 :       z^{M12} = mul((x^{M7} - 1^{M8})^{M9}, y^{M10})^{M11};^{M13}
4 :       return add(z^{M14}, y^{M15})^{M16};^{M17}
        }
5 :     if (x^{M18} == 0^{M19})^{M20}
6 :       return 0^{M21};^{M22}
7 :     if (y^{M23} > 0^{M24})^{M25} {
8 :       z^{M31} = mul(x^{M26}, (y^{M27} - 1^{M28})^{M29})^{M30};^{M32}
9 :       return add(z^{M33}, x^{M34})^{M35};^{M36}
        }
10 :    return mul(-x^{M37}, -y^{M38})^{M39};^{M40}
      }
```

where

$\mathtt{A}_0 = \mathcal{A}_e([\![m]\!], bt_m)$; $\mathtt{A}_1 = \mathcal{A}_e([\![n]\!], bt_n)$; $\mathtt{A}_2 = \mathcal{A}_e([\![m]\!], bt_m)$;
$\mathtt{A}_3 = \mathcal{A}_e([\![n]\!], bt_n)$; $\mathtt{A}_4 = \mathtt{A}_5 = \mathcal{A}_e([\![m + n]\!], bt_m \sqcup bt_n)$;

$\mathtt{M}_0 = \mathcal{A}_e([\![x]\!], bt_x)$; $\mathtt{M}_1 = \mathcal{A}_e([\![y]\!], bt_y)$; $\mathtt{M}_2 = \mathtt{M}_3 = \mathcal{A}_e([\![z]\!], bt_z)$
$\mathtt{M}_4 = \mathtt{M}_5 = \mathtt{M}_6 = \mathcal{A}_e([\![x]\!], bt_x)$; $\mathtt{M}_7 = \mathtt{M}_8 = \mathtt{M}_9 = \mathcal{A}_e([\![x]\!], bt_x)$;
$\mathtt{M}_{10} = \mathcal{A}_e([\![y]\!], bt_y)$; $\mathtt{M}_{11} = \mathcal{A}_e([\![\mathtt{mul}(x - 1, y)]\!], bt_x \sqcup bt_y)$;
$\mathtt{M}_{12} = \mathcal{A}_e([\![z]\!], bt_x \sqcup bt_y)$; $\mathtt{M}_{13} = \mathcal{A}_s([\![z = \mathtt{mul}...]\!], \{\mathtt{M}_{11}, \mathtt{M}_{12}\})$;
$\mathtt{M}_{14} = \mathcal{A}_e([\![z]\!], bt_x \sqcup bt_y)$; $\mathtt{M}_{15} = \mathcal{A}_e([\![y]\!], bt_y)$;
$\mathtt{M}_{16} = \mathtt{M}_{17} = \mathcal{A}_e([\![\mathtt{return}...]\!], bt_x \sqcup bt_y)$;
$\mathtt{M}_{18} = \mathtt{M}_{19} = \mathtt{M}_{20} = \mathcal{A}_e([\![x]\!], bt_x)$; $\mathtt{M}_{21} = \mathtt{M}_{22} = \mathcal{A}_e([\![\mathtt{return}...]\!], bt_x)$
$\mathtt{M}_{23} = \mathtt{M}_{24} = \mathtt{M}_{25} = \mathcal{A}_e([\![y]\!], bt_y)$; $\mathtt{M}_{26} = \mathcal{A}_e([\![x]\!], bt_x)$
$\mathtt{M}_{27} = \mathtt{M}_{28} = \mathtt{M}_{29} = \mathcal{A}_e([\![y]\!], bt_y)$; $\mathtt{M}_{30} = \mathcal{A}_e([\![\mathtt{mul}...]\!], bt_x \sqcup bt_y)$
$\mathtt{M}_{31} = \mathcal{A}_e([\![z]\!], bt_x \sqcup bt_y)$; $\mathtt{M}_{32} = \mathcal{A}_s([\![z = \mathtt{mul}...]\!], \{\mathtt{M}_{30}]\!], \mathtt{M}_{31}\})$
$\mathtt{M}_{33} = \mathcal{A}_e([\![z]\!], bt_x \sqcup bt_y)$; $\mathtt{M}_{34} = \mathcal{A}_e([\![x]\!], bt_x)$;
$\mathtt{M}_{35} = \mathtt{M}_{36} = \mathcal{A}_e([\![\mathtt{add}...]\!], bt_x \sqcup bt_y)$; $\mathtt{M}_{37} = \mathcal{A}_e([\![-x]\!], bt_x)$
$\mathtt{M}_{38} = \mathcal{A}_e([\![-y]\!], bt_y)$; $\mathtt{M}_{39} = \mathtt{M}_{40} = \mathcal{A}_e([\![\mathtt{mul}(-x, -y)]\!], bt_x \sqcup bt_y)$

Figure 7. Action-variable Annotated Code for `add` and `mul` Component

- A unique action value sequence and
- An equivalence set of binding-time signatures, all of which can produce the same action value sequence by the instantiating function **Inst**.

Figure 9 provides an example showing the shared action-annotated code for the `mul` component. For ease of presentation we replace each statement in the component by a number representing its program point (Pp). The second column shows the associated action-variable sequence of each program point. The third column shows a set of pairs generated by the **FindEqSet** method. Lastly, the set of profitable binding-time signatures for function `mul` is:

$$\begin{array}{ll} \mathrm{S}_1: & bt_x = \mathtt{S} \wedge bt_y = \mathtt{S} \\ \mathrm{S}_2: & bt_x = \mathtt{S} \wedge bt_y = \mathtt{D} \\ \mathrm{S}_3: & bt_x = \mathtt{D} \wedge bt_y = \mathtt{S} \end{array}$$

Code sharing for the PSC of the `mul` component can be found clearly in Figure 9. For instance, signatures S_1 and S_2 are detected as equivalent at more than half of their codes, i.e at program points: 1, 2, 4, 5, 6 and 9. Thus the action-annotated codes at those program points are sharable.

Implementation-wise, the set of pairs will be organized into a *web of action-annotated code segments* so that retrieval of a specialized code instance can be performed by traversing through the web.

Method:

$Result$ = Nil;
for (i=0; i <| $\overline{\overline{A_{exp}}}$ |; i++) {
  ($\overline{\overline{A_{val}}}$, EQ_{bts}) = (Nil, Nil);
  for (j=0; j <| $\overline{\sigma}$ |; j++) {
    $\overline{A_{val}}$ = **Inst** ($\overline{A_{exp}}$.i, $\overline{\sigma}$.j);
    if $\overline{A_{val}} \notin \overline{\overline{A_{val}}}$
    then { $\overline{\overline{A_{val}}}$ = $\overline{\overline{A_{val}}}$ ++ [$\overline{A_{val}}$] ;
      EQ_{bts} = EQ_{bts} ++ [[$\overline{\sigma}$.j]]; }
    else { m = **EleAt** ($\overline{\overline{A_{val}}}$, $\overline{A_{val}}$) ;
      EQ_{bts}.m = EQ_{bts}.m ++ [$\overline{\sigma}$.j] ; }
  }
  $Result$ = $Result$ ++ [[($\overline{\overline{A_{val}}}$, EQ_{bts})]];
}
return $Result$;

USING:
$\overline{A_{val}}$: action value sequence;
$\overline{\overline{A_{val}}}$: sequence of action value sequences
EleAt (`L`, `e`): returns the position of an element `e` in a sequence `L`

Figure 8. The **FindEqSet** Method

Pp	Action variable Seq	Equivalence set
1	[$\mathtt{M}_2$, $\mathtt{M}_3$]	{([id, id], [S_1, S_2, S_3])}
2	[$\mathtt{M}_4$, $\mathtt{M}_5$, $\mathtt{M}_6$]	{([ev, ev, ev], [S_1, S_2]), ([id, id, id], [S_3])}
3	[$\mathtt{M}_7$, $\mathtt{M}_8$, $\mathtt{M}_9$, $\mathtt{M}_{10}$, $\mathtt{M}_{11}$, $\mathtt{M}_{12}$, $\mathtt{M}_{13}$]	{([ev, ev, ev, ev, ev, ev, ev], [S_1]), ([ev, ev, ev, id, rc, id, rc], [S_2]), ([id, id, id, ev, rc, id, rc], [S_3])}
4	[$\mathtt{M}_{14}$, $\mathtt{M}_{15}$, $\mathtt{M}_{16}$, $\mathtt{M}_{17}$]	{([id, id, id, id], [S_1, S_2, S_3])}
5	[$\mathtt{M}_{18}$, $\mathtt{M}_{19}$, $\mathtt{M}_{20}$]	{([ev, ev, ev], [S_1, S_2]), ([id, id, id], [S_3])}
6	[$\mathtt{M}_{21}$, $\mathtt{M}_{22}$]	{([ev, ev], [S_1, S_2]), ([id, id], [S_3])}
7	[$\mathtt{M}_{23}$, $\mathtt{M}_{24}$, $\mathtt{M}_{25}$]	{([ev, ev, ev], [S_1]), ([id, id, id], [S_2, S_3])}
8	[$\mathtt{M}_{26}$, $\mathtt{M}_{27}$, $\mathtt{M}_{28}$, $\mathtt{M}_{29}$, $\mathtt{M}_{30}$, $\mathtt{M}_{31}$, $\mathtt{M}_{32}$]	{([ev, ev, ev, ev, ev, ev, ev], [S_1]), ([ev, id, id, id, rc, id, rc], [S_2]), ([id, ev, ev, ev, rc, id, rc], [S_3])}
9	[$\mathtt{M}_{33}$, $\mathtt{M}_{34}$, $\mathtt{M}_{35}$, $\mathtt{M}_{36}$]	{([id, id, id, id], [S_1, S_2, S_3])}
10	[$\mathtt{M}_{37}$, $\mathtt{M}_{38}$, $\mathtt{M}_{39}$, $\mathtt{M}_{40}$]	{([ev, ev, ev, ev], [S_1]), ([ev, id, rc, rc], [S_2]), ([id, ev, rc, rc], [S_3])}

Figure 9. Shared Action-annotated Code for `mul` Component

The results of PSC construction are called pre-GEs, to signify that they are not full generating extensions, and further specialization is required (at application level) to obtain the desired generating extension.

4.3 Wrapped PSC

Now that we have constructed PSC, we need to prepare it for deployment in applications. Specifically, we expect PSC to inter-

act with the specialization engine used at the application level to achieve application-driven specialization.

We intend to use a variant of Tempo [9] as the specialization engine at application level. Specifically, we expect run-time specialization to be employed. This means that generating extensions of applications will be produced during specialization, based on binding-time information about initial inputs to applications.

While both binding-time analysis and action analysis will continue to be performed at application level in the Tempo-variant, they differ from the original analyses in that they can skip their tasks during their interactions with PSC. Instead, such information has already been computed at component level, and it is only required to be packaged and returned to the analyses.

To this end, we wrap a PSC with an interface, and call this embellished component *wrapped PSC*. The interface serves two purposes:

1. The wrapped PSC provides an API for application level binding-time analysis. Whenever the binding-time analysis requires binding-time information from a call to the wrapped PSC, the respective API method will be invoked. This will then query the inner PSC for the binding-time result of the call.

   Contrary to the usual practice in binding-time analysis, wrapped PSC does *not* attempt to generate new binding-time signatures in response to new specialization context. Instead, it will choose the most appropriate binding-time signature (i.e., minimal profitable context) to use in place of the specialization context, adhering to our specialization policy described in Section 3.2. This is because wrapped PSC has captured *all* profitable binding-time signatures, and we can safely deem other binding-time signatures to be *unprofitable* in leading to a profitable specialization.

2. The wrapped PSC provides an API for application-level action analysis. Similar to the API for binding-time analysis, it will be invoked when there is a request for action analysis information of the component function. However, in this case, the API will construct an action-annotated code instance by traversing through the web of code segments, directed by the specific minimal profitable context of the call. Eventually, the API returns the constructed code instance. Joining these instances with the other results from action analysis, we obtain a complete action-annotated tree for the entire application.

Finally, application-level run-time specialization takes place to convert action-annotated trees to ultimate generating extensions, and our specialization component has entirely immersed in the application.

5. Related works

Schultz in his position paper [22] validated the rationale for independent component specialization, which he termed as "black box program specialization". He proposed that component developers be responsible for identifying specialization opportunities without taking into consideration component use contexts. He further proposed the release of specialization opportunities as an abstract specialization interface to the application developers.

In [11, 25], Consel et al. introduced a notion of *specialization class* as a language extension for object-oriented languages, aiming at expressing program specialization (not just specialization opportunities) in a separate and declarative way. Multiple specialization classes can be attached to a single, regular class, capturing different opportunities for specialization.

In the imperative front, Le Meur et al. proposed a *specialization scenario* approach [18, 10] which allows programmers to manually express their desired degree of optimization. This is done by declaring the expected specialization contexts for C function definitions in terms of "external" specialization parameters, which include functions, global variables and data structure intended to be specialized. A group of specialization scenarios for a component are collected into one file named *specialization module*.

Specialization scenarios are also used by Bobeff and Noyé [4, 5] to declare binding-time information for component specialization, which we have described in detail in Section 1.

Instead of manually declaring specialization opportunities, our profitability signatures are derived automatically through identifying profitability points within a component body. In this work, we choose to identify test constructs occurring in if- and while-constructs as a source for profitable specialization. This technique resembles that proposed by Mock [20] for his profiling tool Calpa. Calpa automatically generates declarations for the specializer Dyc for C programs. As Calpa handles C programs rather than components, it is able to perform dynamic analysis over a program to discover the source for specialization opportunity. Doing so for a component could be difficult, since analyzing an application, which embeds a component, will only uncover part of the specialization opportunity of the component.

Compared to the literatures on the declaration of specialization intent, there is relative less work on how to better manage the multiplicity of specialized codes. As mentioned in Section 1, Bobeff et al. organized specialized codes through the idea of *component generators*, they do not inhibit duplication of codes [5].

In [24, 1, 23], the authors adopted an aspect-oriented approach in managing specialized codes. Specifically, aspects are used to keep specialized codes, and the technique of weaving is used to direct function calls to calls for specialized functions.

In this work, we deal with the issue of specialized-code duplication head-on, and aggressively enable sharing among specialized codes. To this ends, we partition a code into code segments, and maintain a web of code segments. Consequently, we do not produce generating extensions at the end of component specialization. Instead, we provide sufficient information in the PSC so that generating extensions can be quickly assembled by the wrapped PSC during application specialization.

6. Conclusion

Independent component specialization gives rise to new challenges in the specialization research community. It requires specialization to be conducted without initial binding-time contexts. It also requires careful management (especially in space) of multiple specialized codes in order for the specialization component to be receptive to vastly different use contexts in different applications.

In this paper, we proposed two novel techniques to address these new challenges: We advocate the discovery of specialization opportunity by examining the body of the component, and introduce the notion of "profitability declaration" to capture specialization opportunities independent of how components are deployed. Thus the focus of specialization is shifted from the propagation of initial specialization context to the exploration of profitable specialization opportunity. Profitability analysis automatically converts the conceptual profitability declarations to the desired profitability signatures. Furthermore, we propose the construction of code-sharing PSC to minimize the amount of code redundancy arisen from having to maintain specialized codes for diverse use contexts.

We have developed a prototype implementing the profitability analysis and PSC construction. Our preliminary investigation supports the feasibility of this approach, and we are currently subjecting this framework to more experiments. We also intend to refine our framework, especially in the construction of PSC: This includes allowing the sharing of action-annotated codes beyond statement level. Specifically, a set of adjacent statements that share the same

equivalent set of binding-time signatures can be grouped to form a bigger block of code.

Acknowledgements

We would like to thank the anonymous referees for their insightful and detailed comments. We are very grateful to Julia Lawall, Ulrik Pagh Schultz, Charles Consel and Anne-Françoise Le Meur for their professional comments regarding this research through personal communications with the first author and their kindness in sharing their experiences in using Tempo system [15]. This research is partially supported by the university research grant R-252-000-138-112.

References

[1] Helle Markmann Andersen and Ulrik Pagh Schultz. Declarative specialization for object-oriented-program specialization. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 27–38, Verona, Italy, 2004.

[2] Peter Holst Andersen and Carsten Kehler Holst. Termination analysis for offline partial evaluation of a higher order functional language. In *International Symposium on Static Analysis*, pages 67–82, Aachen, Germany, 1996.

[3] Sapan Bhatia, Charles Consel, and Calton Pu. Remote customization of systems code for embedded devices. In *EMSOFT '04: Proceedings of the fourth ACM international conference on Embedded software*, pages 7–15, Pisa, Italy, 2004.

[4] Gustavo Bobeff and Jacques Noyé. Molding components using program specialization techniques. In *Workshop on Component-Oriented Programming*, Darmstadt, Germany, July 2003.

[5] Gustavo Bobeff and Jacques Noyé. Component specialization. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 39–50, Verona, Italy, 2004.

[6] Anders Bondorf. Automatic autoprojection of higher order recursive equations. In *European Symposium on Programming*, pages 70–87, Copenhagen, Denmark, 1990.

[7] Charles 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, Copenhagen, Denmark, 1993.

[8] Charles Consel and Olivier Danvy. From interpreting to compiling binding times. In *European Symposium on Programming*, pages 88–105, Copenhagen, Denmark, 1990.

[9] Charles Consel, Luke Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, Julia Lawall, and J. Noyé. Tempo: specializing systems applications and beyond. *ACM Computing Survey*, 30(3es):19–24, 1998.

[10] Charles Consel, Julia Lawall, and Anne-Françoise Le Meur. A tour of Tempo: A program specializer for the C language. *Science of Computer Programming*, 2004.

[11] Crispin Cowan, Andrew Black, Charles Krasic, Calton Pu, Jonathan Walpole, Charles Consel, and Eugen-Nicolae Volanschi. Specialization classes: an object framework for specialization. In *IWOOOS '96: Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, page 72, Washington, DC, USA, 1996. IEEE Computer Society.

[12] Jeffrey Dean, Craig Chambers, and David Grove. Identifying profitable specialization in object-oriented languages. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–96, Orlando, Florida, 1994.

[13] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 93–102, La Jolla, California, United States, 1995.

[14] Arne John Glenstrup and Neil D. Jones. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM Trans. Program. Lang. Syst.*, 27(6):1147–1215, 2005.

[15] Phoenix group. Tempo Specializer - A Partial Evaluator for C. URL: http://phoenix.labri.fr/software/tempo/.

[16] George T. Heineman and William T. Councill. *Component-Based Software Engineering : Putting the Pieces Together*. Addison-Wesley, 2001.

[17] Neil D. Jones, Cartesten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993.

[18] Anne-Françoise Le Meur, Julia Lawall, and Charles Consel. Specialization scenarios: A pragmatic approach to declaring program specialization. *Higher-Order and Symbolic Computation*, 17(1):47–92, 2004.

[19] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–52, Oct 1992.

[20] Markus Mock. *Automating Selective Dynamic Compilation*. PhD thesis, Department of Computer Science & Engineering, University of Washington, August 2002.

[21] Jean-Guy Schneider and Jun Han. Components: the past, the present, and the future. In *Workshop on Component-Oriented Programming*, Oslo, Norway, 2004.

[22] Ulrik Pagh Schultz. Black-box program specialization. In *Workshop on Component-Oriented Programming*, Lisbon, Portugal, 1999.

[23] Ulrik Pagh Schultz. Private communication, August 2005.

[24] Ulrik Pagh Schultz, Julia Lawall, and Charles Consel. Automatic program specialization for java. *ACM Transaction on Programming Languages and Systems*, 25(4):452–499, 2003.

[25] Eugen-Nicolae Volanschi, Charles Counsel, Gilles Muller, and Crispin Cowan. Declarative specialization of object-oriented programs. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 286–300, Atlanta, Georgia, United States, 1997.

[26] Ping Zhu and Siau Cheng Khoo. Request and assert: A pragmatic approach to generating specialization scenarios. Technical report, School of Computing, National University of Singapore, 2006.