SPECIALIZATION OF APPLICATIONS USING SHARED LIBRARIES

ZHU Ping

(M.Eng, Nanjing University, China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

June 2009

I dedicate this dissertation to my heartwarming family: my wife Wei, for her precious love, encouragement and patience; my parents and elder brother, for their countless support and confidence in me.

ACKNOWLEDGEMENTS

In the acknowledgements, I would express my gratitude to all the people who have assisted me a lot in every aspects during the last seven years.

First of all, I would like to thank my advisor, Prof. KHOO Siau-Cheng. He introduced me to the fancy field of partial evaluation. During the last six years, his constant encouragement, thoughtful insight and careful guidance have brought my research to fruition. He actively and consciously cultivated my research methodology and skills by teaching me about writing papers, giving presentations, and doing independent research. At the final stage of my PhD study, he gave me numerous supports and constructive suggestions regarding my future career. Besides being a nice advisor on my research-related matters, he often shared with me his positive life wisdom when I was psychologically low on non-research related matters. To him, I express my deepest gratitude.

I am indebted to Prof. Julia Lawall, who always responded me timely regarding my questions about Tempo, carefully read and made detailed corrections on my manuscripts; to Prof. Neil Jones, who gave me a lot of critical and constructive comments on my dissertation, and taught me a lot about Futamura Projection during his stay in Singapore from January 2008 to March 2008; to Dr. Hugh Anderson, who provided me professional comments on my research and helped me a lot in Linux-programming and proof reading of my manuscripts.

I appreciate Prof. Ulrik Pagh Schultz, Anne-Françoise Le Meur, Charles Consel, Craig Chambers, Dr. Sapan Bhatia, Briant Grant for their help in understanding the details of Tempo and DyC.

I sincerely thank my friends WANG Meng, Kenny LU Zhuo-Ming, WANG Tao,

SHAO Xi, CHENG Da-Ming, PAN Yu, HU An, JI Li-Ping, LIANG Hui, CAO Dong-Ni, CHENG Chun-Qing, XU Xin, HE Xiao-Li, Dana XU Na, LIU Zeng-Jiao, QIN Sheng-Chao and SOH Jen, for their friendship and accompany during my stay in Singapore. They keep me updated of issues outside partial evaluation and make my life amiable.

I am grateful to the members of the Programming Language and System Laboratory II, who provided useful feedback and advice in my presentation rehearsals. Special thanks go to Prof. CHIN Wei-Ngan, Martin Sulzmann, Andrei Stefan, Nguyen Huu-Hai, Florin Craciun, Corneliu Popeea, David Lo, Beatrice Luca, Cristina David, LAM Edmund Soon Lee, for provocative conversations and cheerful parties.

Finally, my acknowledgements go to National University of Singapore for generously providing me research scholarship necessary to pursue this work from July 2002 to July 2006; and travel grants to cover the expenses incurred by attending several leading academic conferences abroad. My thanks also go to the administrative and technical support staff in the School of Computing, especially Ms. LOO Laifeng, HEE Tse Wei Emily; to Ms Virginia De Souza who provided me a professional personal consultant service; to those cleaners who daily keep the lab clean. Their support and service are more than what I have expected.

TABLE OF CONTENTS

ABSTRACT

In the last decade, **shared libraries** have became popular commodities for implementing essential services in many systems and application domains. The prevalence of shared libraries depends on not only their support for software reuse, but also their allowance for **sharing** at both compile-time and run-time.

On the other hand, the reuse of libraries results in degradation of system performance, primarily due to the adaption of the general-purpose libraries to the specific contexts when they are deployed in various applications. To reconcile the conflicting requirements of generality of shared libraries across all applications and high performance for individual applications, shared libraries are subject to specialization.

This dissertation introduces a comprehensive framework for **specialization of applications using shared libraries**. This framework preserves sharing of shared libraries, enables reduction of code duplication during the entire specialization process, and enhances existing specialization techniques through cross-fertilization between program slicing and partial evaluation.

Technically, we introduce a **profitability analysis** aiming at discovering all meaningful specialization opportunities of a shared library without taking into consideration its deployment context. We propose methodologies for constructing and executing a **generic specialization component** for a shared library catering to various specialization opportunities. These methodologies enable code/memory reduction at compile-time and run-time through sharing. Finally, we investigate the **essence and uniformity of program slicing and partial evaluation**. The uniformity enables cross-fertilization between program slicing and partial evaluation such that existing specialization techniques can be enhanced.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Shared Libraries

A library is a general-purpose program (existing in source form or binary form) which can be reused to develop various applications. Libraries are commonly categorized into two types, namely **static libraries** and **shared libraries**, according to the ways they are linked with applications. The binaries of static libraries are copied into the binary of an application at link-time to produce a stand-alone executable. On the other hand, the binaries of shared libraries are only loaded into memory to execute an application at load-time or run-time. In the last decade, shared libraries are becoming popular commodities for implementing essential services in many systems and application domains. For example, in the Windows system, many device drivers and resource files are presented in the form of dynamically linked libraries, which are Microsoft's implementation of the shared libraries.

The prevalence of shared libraries depends on not only their support for software reuse, but also their allowance for **sharing**; ie. (1) There is only one copy of a shared library's binary on the disk. The binary of an application that uses one or more shared libraries, contains only references to the binaries of those shared libraries, and (2) At run-time there is one single copy of the binary of a shared library in memory. The executions of all applications that use the shared library refers to the same copy of the binary of the shared library.

Overall, sharing aims at reducing code duplication and achieving reduction in both disk and memory use. Furthermore, it enables transparent updating, i.e. all applications that use the shared library immediately enjoy the bug fixing for that

shared library without having to be rebuilt since only one copy of the shared library is maintained.

1.2 Program Specialization

The reuse of libraries results in degradation of system performance, primarily due to the adaption of the general-purpose libraries to the specific contexts when they are used in various applications. This degradation has been recognized in many areas such as operating systems and graphics. There are two common sources of context-related inefficiencies of a library. The first is the presence of useless computations in a library when the library is used to solve a specific problem. The second is the presence of partial inputs to a library that do not change very often but nevertheless cause the libraries to repeatedly perform the computations dependent on this invariant information.

To reconcile the conflicting requirements of generality of a library across all applications and high performance for individual applications, libraries are subject to specialization. There have been many well-developed program specialization tools used to tackle these two common inefficiencies. Program slicing, which was first introduced as a debugging technique, can also be used to perform a kind of program specialization, as argued by Reps in [67], by extracting from the original program a semantics-preserving sub-program confined to a specific application. On the other hand, partial evaluation specializes a program with respect to its invariant partial input and produces a more efficient specialized program at compile-time.

We term the information that stipulates the context in which the program could be specialized as **specialization information**. Furthermore, we term the program transformer, such as a program slicer or a partial evaluator, which defines a set of transformation rules and transforms the original program into a specialized program

materializing the specialization opportunities specified in the specialization information as a **specialization engine**.

1.3 Specialization of Applications Using Shared Libraries

The importance of specialization of applications using libraries has been recognized by the partial evaluation community and substantial progress has been made over the past several years to make partial evaluation come true in practice. Tempo, which is a successful partial evaluator for C language, advocates *modular specialization* as explained in [1],

> "... It is not usually practical or even desirable to apply specialization to a complete application, i.e. from the main function down to all leaf functions. Instead, specialization is usually applied to part of an application (without altering the rest of the application) or to library functions. Modular specialization supports specializing a fragment of an application ... "

The **specialization of applications using shared libraries** to be studied in this dissertation is different from conventional program specialization techniques, which have been designed for specializing applications using static libraries. It has the intention of preserving *sharing* during the entire specialization process, from specializing shared libraries at compile-time to executing specialized applications that use the specialized shared libraries at run-time. Correspondingly, specialization of applications using shared libraries can be divided into the following sub-problems.

The first sub-problem is called **independent specialization information generation**. The first step to ensure that specialization preserves sharing is to enable independent specialization of shared libraries, i.e., shared libraries are specialized

independently, free from their deployment contexts confined to any specific applications. The focus of specialization is *how best to prepare a library for specialization such that the specialized library remains effective in as many applications as possible.* The specialization information can be abstracted from the context in which a library interacts with other libraries or derived from the specialization opportunities residing inside the library. The latter approach enables library developers to take advantage of their knowledge of a library's implementation and prepare suitable specialization information for all possible future deployments.

The second sub-problem is called **efficient specialized library construction and execution**. The original libraries will be replaced by their corresponding specialized libraries for specialization purpose that cater for various specialization opportunities. In this way, we minimize the need for repetitive and redundant specialization of libraries at the application level. Given that normally several pieces of specialization information are produced in independent specialization information generation, it becomes important to manage and balance the trade-off between the *multiplicity* of specialized libraries generated with respect to those various pieces of specialization information, and the space required for keeping them, which demonstrates the sharing property. In principle, we would like to be able to generate these specialized libraries at compile-time, in order to enable maximal sharing before deploying them in multiple applications. It is also desirable to exploit the specialized libraries at run-time to minimize the footprints produced from them.

The third sub-problem is called **specialization engine enhancement**. The specialization of applications using shared libraries leverages on the maturity of existing implementations of specialization techniques, in particular partial evaluation, that have been under development for several years. It is desirable to enhance existing specialization techniques through cross-fertilization among different specialization techniques. Typically, partial evaluation has been used in exploiting **requirements**,

which constrain the kinds of *input* permissible for invoking a library. To specialize a library, partial evaluation propagates invariant input information *forward* to the library's output. On the other hand, program slicing has been used to specialize a library with respect to **assertions**, which stipulate the kind of *output* behavior acceptable by the calling context. Program slicing performs *backward* specialization which passes information from output back to the library's input. Given the intimate relation between the requirement and the assertion of a library as advocated by the *design by contract* methodology [59], it is natural to study the relation between partial evaluation and program slicing, and to explore their potential for improving upon the existing specialization techniques.

1.4 Contributions

In this dissertation, we conduct a comprehensive study of specialization of applications using shared libraries. Our goal is to develop a framework that *preserves sharing of shared libraries, reduce code duplication during the entire process of specializing applications using shared libraries through sharing, and enhances existing specialization techniques through cross-fertilization between program slicing and partial evaluation*. The technical contributions of this dissertation can be summarized as follows.

- To address the first sub-problem of **independent specialization information generation**, we design a **profitability analysis** aiming at discovering all meaningful specialization information of a shared library without taking its deployment context into consideration. Specifically, we advocate **profitability declaration**, a novel methodology to capture specialization opportunities inside a library. This conceptual profitability declaration is translated into a **profitability signature** which is expressed in the form of a binding-time constraint. A profitability signature stipulates a constraint enforced over library

parameters in order to materialize the specialization opportunities within a library.

- To address the second sub-problem of **efficient specialized library construction and execution**, we propose methodologies to construct and execute a **generic specialization component** (GSC for short) for a shared library. A GSC caters for the various specialization opportunities of a library returned by profitability analysis. These methodologies enable reduction of duplicated code at both compile-time and run-time. Technically, we design a static transformation to detect sharable templates and eliminate duplicated templates when constructing a GSC for a library at compile-time. We adopt a strategy for template dumping that minimizes the footprints of shared libraries in the specialized applications by reducing the number of duplicated object templates created in a dynamically allocated memory region at run-time. With this new strategy, we propose a run-time specialization mechanism to manage the new structure of the footprint.

- To address the third sub-problem of **specialization engine enhancement**, we build a theoretical framework which captures the essence and uniformity of program slicing and partial evaluation. The uniformity between these two techniques enables cross-fertilization between slicing and partial evaluation to enhance existing specialization techniques.

1.5 Organization of the Dissertation

The rest of the dissertation is organized as follows. The next chapter introduces background on partial evaluation and program slicing to facilitate understanding the technical details of our approach. Chapter 2 also presents an overview of the approach taken in this dissertation. Chapter 3 surveys related research published in this domain

prior to and during the course of our work. Chapter 4 to Chapter 6 describe in detail the contributions of this dissertation. In Chapter 4 profitability analysis is introduced along with a profitability-oriented binding-time analysis. This chapter is largely based on the work done in 2006 and 2007, and reported in [82, 83, 84], but extended here for clarification. Chapter 5 presents the approach to efficiently constructing and executing a generic specialization component for a library. This topic was covered in [85], but is again clarified and extended in this dissertation. Chapter 6 presents a theoretical unified framework in which we can cast both (forward and backward) program slicing and partial evaluation, and develop a new specialization framework that provides for cross-fertilization between existing program slicing and partial evaluation techniques. This work was reported in [81]. Finally, Chapter 7 summarizes the contribution of the research and points out possible future directions.

1.6 Notational Conventions

The notation and font styles used throughout this dissertation are defined as follows.

- The generic entities and concrete entities (including constant values or plain program fragments) are written math font and teletype font respectively. For instance, in "$v = e$" v ranges over all variables and e ranges over all expressions; in "`v=v+1`" the LHS is the concrete variable `v` and `1` is a constant value.

- The name of a type is written in bold font and its initial letter is capitalized. A value x (either a data or a function) of type $\mathbf{T}$ is written as $x \in \mathbf{T}$.

- The notation $\sigma[x \leftarrow new_x]$ represents an updating function which gets the primary index x of the host data structure σ mapped to its new value new_x.

- The notation $[\![p]\!]$ represents a semantic function of the underlying programming language in which the code p is written.

- Notation for some data structures: where $ele_1, \ \ldots, \ ele_n$ are elements of those data structures
  - A set: $\{ele_1, \ \ldots, \ ele_n\}$
  - A list: $[ele_1, \ \ldots, \ ele_n]$

    A stack, which is a last-in-first-out list has the same representation as that of a list with an extra requirement that the elements pushed into the stack earlier will be at right hand side of the elements pushed into the stack later.
  - A tuple: $\langle ele_1, \ \ldots, \ ele_n \rangle$
  - A record: $\langle \mathbf{fld_1} : ele_1, \ \ldots, \ \mathbf{fld_n} : ele_n \rangle$ where $\{\mathbf{fld_i}\}$ denote field names. For the ease of presentation, the field names $\{\mathbf{fld_i}\}$ of a record are omitted in descriptions or algorithms that refer to the record.

CHAPTER 2

OVERVIEW

In this chapter, we first introduce the subject language used in this dissertation. Then we give the background information on program slicing and partial evaluation to facilitate understanding the technical details of our approaches elaborated in Chapters 4, 5 and 6. Finally we present our overall approaches to addressing the three sub-problems: independent specialization information generation, efficient specialized library construction and execution, and specialization engine enhancement, as stated in Section 1.3.

2.1 Language

In this dissertation we choose a shared library to be a function definition, which may be interrelated with other function definition.[1] The terms **shared library** and **function definition** are treated as synonyms and used interchangeably in the remaining part of the dissertation. The subject language is a subset of the C language and its abstract syntax is defined in Figure 2.1.

The evaluation strategy of library calls is limited to call-by-value and every library must return a value. We adopt an assumption that works well in practice that the return value of a library must be (data- and control-) dependent on *all* the library's parameters. We also assume that all the programs written in this subject language terminate.

Figure 2.2 presents a self-recursive library `power`, which computes the base `b` to the power `e`.

[1]Each file or module contains only one function definition.

c	$\in$	**Const**	Numerals or Booleans
v	$\in$	**Var**	Variables
f, g	$\in$	**FName**	Library names
b_{op}	$\in$	**BOp**	Binary operators
	::=	$+ \mid - \mid * \mid / \mid == \mid != \mid$	
		$< \mid > \mid >= \mid <= \mid \&\& \mid \mid\mid$	
e	$\in$	**Exp**	Expressions
	::=	$c \mid v \mid f\ (e_1, \ldots, e_n) \mid e_1\ b_{op}\ e_2$	
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
	::=	$\texttt{int}\ v$	
$locals$	$\in$	**Locals**	Local variable declarations
	::=	$decl;$	
$paras$	$\in$	**Paras**	Parameter list
	::=	$decl \mid decl, paras$	
fd	$\in$	**FDef**	Libraries
	::=	$\texttt{int}\ f\ (paras)\ \{locals^*\quad s\}$	

Figure 2.1: Syntax of the subject language

2.2 Background on Program Slicing

Program slicing, which was first introduced by Mark Weiser [80] as a debugging technique, is a decomposition technique that extracts from an original program those statements relevant to a particular computation. He defined that a program p' is a slice of an original program p if p' is a syntactic subset of p and p' is guaranteed to faithfully represent p within the domain of specified subset of behavior, which is referred to as a **slicing criterion**. A complete survey on program slicing can be found in [76].

```
int power (int b, int e) {
  int z;
  if (e == 0)
    return 1;
  else {
    z = b * power(b, e-1);
    return z;
  }
}
```

Figure 2.2: Library `power`

Weiser's program slicing has been known as static slicing, because the slicing criterion contains no information about how the program is executed. For static slicing, the slicing criterion is encoded as a pair $\langle pp,\ V \rangle$ where pp is a program point and V is an arbitrary set of variables appearing at program point pp.

A statement is included in a slice when it contains variables whose values are involved either directly or indirectly in the computation of those variables declared in the slicing criterion. We term these variables, including those in the slicing criterion, **residual variables**. On the other hand, variables that cannot be affected by (or affect) the residual variables are termed as **transient variables**. Note that such a classification of variables is dependent on the program point. A variable may be transient at one program point, and residual at another.

Normally, static slicing can be categorized into **forward static slicing** and **backward static slicing**. Forward static slicing of a program simply extracts those statements and/or predicates in the program that are affected by the slicing criterion. On the other hand, backward static slicing extracts those statements and/or predicates that can have effect on the slicing criterion.

2.3 Background on Partial Evaluation

Jones [46] defined partial evaluation as a two-stage computation. In stage one, a partial evaluator mix specializes a source program p with respect to invariant partial input in_1, and produces the specialized code p_{in_1}. In stage two, p_{in_1} is executed on the remaining input in_2 to produce the same result out as running the source program p on all of its input, provided that all the computations involved terminate. Formally,

Computations in stage one:	$p_{in_1} = [\![mix]\!]\ [p, in_1]$
Computations in stage two:	$out = [\![p_{in_1}]\!]\ in_2$
An equational definition of mix:	$[\![p]\!]\ [in_1, in_2] = [\![[\![mix]\!]\ [p, in_1]]\!]\ in_2$

The chief motivation for partial evaluation is speed: program p_{in_1} is often faster than the original program p because it does not need to perform the computations that solely depend on the invariant in_1.

2.3.1 Offline Partial Evaluation

Partial evaluation transforms program statements in two ways: It either reduces (a.k.a evaluates) a program construct (an expression or a statement) whose computation is solely based on invariant partial input, but keeps its effect within a partial evaluation environment; or residualizes the program construct whose computation relies on varying input to form the specialized program.

According to how the transformation decisions are made, partial evaluation is normally categorized into **online partial evaluation** and **offline partial evaluation**. Online partial evaluation determines and performs the transformations in a single pass in the presence of the concrete values of the invariant inputs. On the contrary, offline partial evaluation typically involves a preprocessing phase called **binding-time analysis** (BTA for short), in which the transformation decisions are made.

BTA attempts to determine at each program point the **binding time** of the syntactic construct, which asserts whether that program construct will be bounded to a concrete value at stage one or stage two, and produces a two-level binding-time annotated program [61]. The input to BTA is termed as a **binding-time division** over program parameters [45], which specifies binding times of program parameters.

The syntax of the binding-time information used in this dissertation is defined in Figure 2.3.

bt_v	$\in$	$\mathbf{BT_v}$	Binding-time variables
bt_e	$\in$	$\mathbf{BT_e}$	Binding-time expressions
	$::=$	$\mathtt{s} \mid \mathtt{d} \mid bt_v \mid bt_{e_1} \sqcup bt_{e_2} \mid bt_{e_1} \sqcap bt_{e_2}$	

Figure 2.3: Syntax of binding-time information

We consider three primitive binding-time expressions: Two binding-time constants static (`s`) and dynamic (`d`) representing that values are bounded to variables at stage one and stage two respectively, and a binding-time variable bt_v ranging over `s` and `d`. `s` and `d` are ordered in decreasing staticness: $\mathtt{s} \sqsubset \mathtt{d}$. 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.

For example, Figure 2.4 depicts a binding-time annotated library `power` produced with respect to a binding-time division ($bt_b = \mathtt{d} \wedge bt_e = \mathtt{s}$), where bt_b and bt_e are binding-time variables pertaining to the library's parameters `b` and `e`, respectively.

Several popular partial evaluators such as Schism [20, 21] and Tempo [21, 22] further employ **action analysis** after BTA to aid specialization. As pointed by Consel et al in [21], the action annotations attached to program constructs are control-based directives to drive the partial evaluators as to what to do for each expression and therefore the partial evaluators are guided first by the action tree and then by the abstract syntax tree of codes – instead of performing first a syntax analysis and

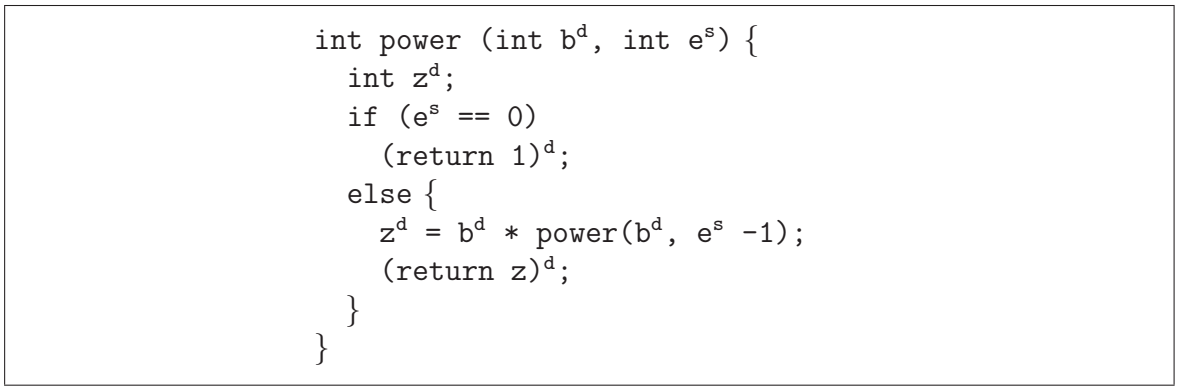

Figure 2.4: A binding-time annotated library `power`

then interpreting binding times. The action annotation domain $\mathbf{AC_{val}}$ comprises four values `ev`, `rd`, `rb`, and `id`, which represent four transformations *evaluate, reduce, rebuild and reproduce*, respectively. The action annotation of each program construct is strictly determined by its binding time. Figure 2.5 depicts an action annotated library `power` produced by Tempo from the binding-time annotated code shown in Figure 2.4. For clarity, we omit those action annotations that can be inferred easily. For example, if an expression or a statement is annotated by `ev` (or `id`), the action annotations of all the nested program constructs will also be `ev` (or `id`) and are thus omitted.

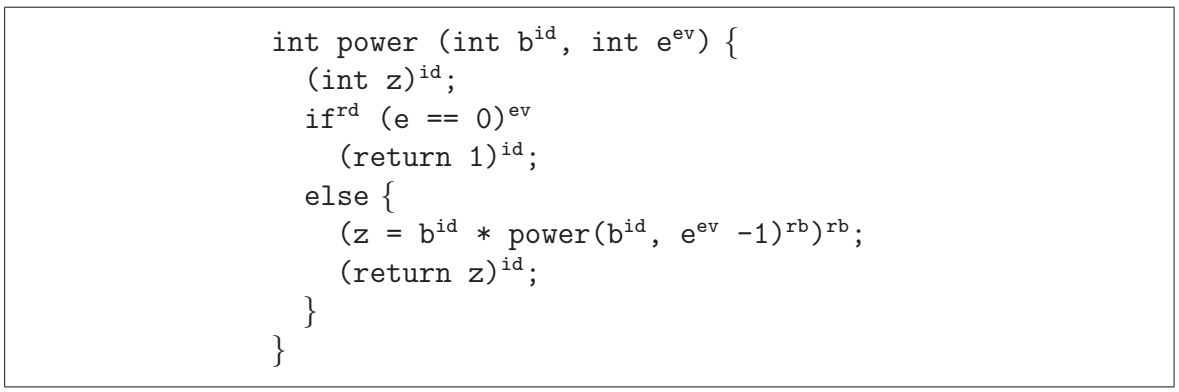

Figure 2.5: An action annotated library `power` produced by Tempo

Figures 2.6, 2.7 and 2.8 define the rules of action computation of program

constructs from the corresponding binding times. $\mathcal{AC}_e$ takes in a binding-time annotated expression (of type $\mathbf{Exp^{bt}}$), and returns an action-annotated expression (of type $\mathbf{Exp^{aa}}$). $\mathcal{AC}_s$ takes in a binding-time annotated statement (of type $\mathbf{Stat^{bt}}$), and returns an action-annotated statement (of type $\mathbf{Stat^{aa}}$). The operation *outmost* extracts the outermost action annotation of an action-annotated expression (or statement). Interested readers may wish to refer to [21] for the original motivation and detailed implementation of action computation for program constructs from the corresponding binding times.

$$
\begin{array}{lcl}
\mathcal{AC}_e & \in & \mathbf{Exp^{bt}} \rightarrow \mathbf{Exp^{aa}} \\
\\
\mathcal{AC}_e\ c^{bt} & ::= & \mathit{if}\ \ (bt == \mathtt{s}) \\
 & & \mathit{then}\ \ c^{\mathtt{ev}} \\
 & & \mathit{else}\ \ c^{\mathtt{id}} \\
\\
\mathcal{AC}_e\ v^{bt} & ::= & \mathit{if}\ \ (bt == \mathtt{s}) \\
 & & \mathit{then}\ \ v^{\mathtt{ev}} \\
 & & \mathit{else}\ \ v^{\mathtt{id}} \\
\\
\mathcal{AC}_e\ (e_1^{\mathbf{bt}}\ b_{op}\ e_2^{\mathbf{bt}}) & ::= & \mathit{let}\ \ e_1^{\mathbf{aa}} = \mathcal{AC}_e\ e_1^{\mathbf{bt}} \\
 & & \quad\ aa_1 = \mathit{outmost}\ \ e_1^{\mathbf{aa}} \\
 & & \quad\ e_2^{\mathbf{aa}} = \mathcal{AC}_e\ e_2^{\mathbf{bt}} \\
 & & \quad\ aa_2 = \mathit{outmost}\ \ e_2^{\mathbf{aa}} \\
 & & \mathit{in}\ \ \mathit{if}\ (aa_1 == \mathtt{ev}) \wedge (aa_2 == \mathtt{ev}) \\
 & & \quad\ \mathit{then}\ (e_1^{\mathbf{aa}}\ b_{op}\ e_2^{\mathbf{aa}})^{\mathtt{ev}} \\
 & & \quad\ \mathit{else\ if}\ (aa_1 == \mathtt{id}) \wedge (aa_2 == \mathtt{id}) \\
 & & \quad\ \mathit{then}\ (e_1^{\mathbf{aa}}\ b_{op}\ e_2^{\mathbf{aa}})^{\mathtt{id}} \\
 & & \quad\ \mathit{else}\ (e_1^{\mathbf{aa}}\ b_{op}\ e_2^{\mathbf{aa}})^{\mathtt{rb}} \\
\\
\mathcal{AC}_e\ f\ (e_1^{\mathbf{bt}}, \ldots, e_n^{\mathbf{bt}})\ \rho\ \tau\ ctr & ::= & \mathit{let}\ \ \{e_i^{\mathbf{aa}} = \mathcal{AC}_e\ e_i^{\mathbf{bt}} \mid 1 \leq i \leq n\} \\
 & & \quad\ \{aa_i = \mathit{outmost}\ \ e_i^{\mathbf{aa}} \mid 1 \leq i \leq n\} \\
 & & \mathit{in}\ \ \mathit{if}\ (aa_1 == \mathtt{ev}) \wedge \ldots \wedge (aa_n == \mathtt{ev}) \\
 & & \quad\ \mathit{then}\ f\ (e_1^{\mathbf{aa}}, \ldots, e_n^{\mathbf{aa}})^{\mathtt{ev}} \\
 & & \quad\ \mathit{else\ if}\ (aa_1 == \mathtt{id}) \wedge \ldots \wedge (aa_n == \mathtt{id}) \\
 & & \quad\ \mathit{then}\ f\ (e_1^{\mathbf{aa}}, \ldots, e_n^{\mathbf{aa}})^{\mathtt{id}} \\
 & & \quad\ \mathit{else}\ f\ (e_1^{\mathbf{aa}}, \ldots, e_n^{\mathbf{aa}})^{\mathtt{rb}}
\end{array}
$$

Figure 2.6: Action analysis over an expression

$$
\begin{array}{lcl}
\mathcal{AC}_s & \in & \mathbf{Stat^{bt}} \rightarrow \mathbf{Stat^{aa}} \\
\\
\mathcal{AC}_s\ (s_1^{\mathbf{bt}}\ ;\ s_2^{\mathbf{bt}}) & ::= & \begin{array}{ll} let & \{s_i^{\mathbf{aa}} = \mathcal{AC}_s\ s_i^{\mathbf{bt}} \mid i = 1,2\} \\ in & (s_1^{\mathbf{aa}}\ ;\ s_2^{\mathbf{aa}}) \end{array} \\
\\
\mathcal{AC}_s\ (v^{bt_1} = e^{\mathbf{bt}}) & ::= & \begin{array}{ll} let & v^{aa_v} = \mathcal{AC}_e\ v^{bt_1} \\ & e^{\mathbf{aa}} = \mathcal{AC}_e\ e^{\mathbf{bt}} \\ & aa_e = outmost\ \ e^{\mathbf{aa}} \\ in & if\ (aa_v == \mathtt{ev}) \wedge (aa_e == \mathtt{ev}) \\ & then\ (v^{aa_v} = e^{\mathbf{aa}})^{\mathtt{ev}} \\ & else\ if\ (aa_v == \mathtt{id}) \wedge (aa_e == \mathtt{id}) \\ & then\ (v^{aa_v} = e^{\mathbf{aa}})^{\mathtt{id}} \\ & else\ (v^{aa_v} = e^{\mathbf{aa}})^{\mathtt{rb}} \end{array} \\
\\
\mathcal{AC}_s\ (\mathtt{int}\ v_1^{bt_1},\ \ldots,\ \mathtt{int}\ v_n^{bt_n}) & ::= & \begin{array}{ll} let & \{v_i^{aa_i} = \mathcal{AC}_e\ v_i^{bt_i} \mid 1 \leq i \leq n\} \\ in & (\mathtt{int}\ v_1^{aa_1},\ \ldots,\ \mathtt{int}\ v_n^{aa_n}) \end{array} \\
\\
\mathcal{AC}_s\ (\mathtt{return}\ e^{\mathbf{bt}}) & ::= & \begin{array}{ll} let & e^{\mathbf{aa}} = \mathcal{AC}_e\ e^{\mathbf{bt}} \\ & aa = outmost\ e^{\mathbf{aa}} \\ in & (\mathtt{return}\ e^{\mathbf{aa}})^{aa} \end{array}
\end{array}
$$

Figure 2.7: Action analysis over a statement: Part 1

2.3.2 Run-time Partial Evaluation

According to when the concrete values of in_1 and in_2 are available, partial evaluation is commonly divided into **compile-time partial evaluation** and **run-time partial evaluation**. In compile-time partial evaluation, concrete values of in_1 and in_2 are available at compile-time and run-time, respectively. For run-time partial evaluation, values of in_1 and in_2 are only known at run-time though still in two stages. Such a situation occurs, for example, when a set of functions implement session-oriented transactions, as noted in [23].

In this dissertation, we do *not* specialize a shared library with respect to concrete values, as it is rare to establish concrete specialization values for an off-the-shelf library and such values are only provided at the application level. Instead, a binding-time

$$
\begin{array}{lll}
\mathcal{AC}_s\ (\texttt{if}\ e^{\mathbf{bt}}\ s_1^{\mathbf{bt}}\ \texttt{else}\ s_2^{\mathbf{bt}}) & ::= & \mathit{let}\ \ e^{\mathbf{aa}} = \mathcal{AC}_e\ e^{\mathbf{bt}} \\
 & & \quad aa_e = \mathit{outmost}\ \ e^{\mathbf{aa}} \\
 & & \quad \{s_i^{\mathbf{aa}} = \mathcal{AC}_s\ s_i^{\mathbf{bt}} \mid i = 1,2\} \\
 & & \quad \{aa_i = \mathit{outmost}\ \ s_i^{\mathbf{aa}} \mid i = 1,2\} \\
 & & \mathit{in}\ \ \mathit{if}\ (aa_e == \texttt{ev}) \wedge (aa_1 == \texttt{ev}) \wedge (aa_2 == \texttt{ev}) \\
 & & \quad \mathit{then}\ \texttt{if}\ e^{\mathbf{aa}}\ s_1^{\mathbf{aa}}\ \texttt{else}\ s_2^{\mathbf{aa}})^{\texttt{ev}} \\
 & & \quad \mathit{else\ if}\ (aa_e == \texttt{id}) \wedge (aa_1 == \texttt{id}) \wedge (aa_2 == \texttt{id}) \\
 & & \quad \mathit{then}\ (\texttt{if}\ e^{\mathbf{aa}}\ s_1^{\mathbf{aa}}\ \texttt{else}\ s_2^{\mathbf{aa}})^{\texttt{id}} \\
 & & \quad \mathit{else\ if}\ (aa_e == \texttt{ev}) \\
 & & \quad \mathit{then}\ (\texttt{if}\ e^{\mathbf{aa}}\ s_1^{\mathbf{aa}}\ \texttt{else}\ s_2^{\mathbf{aa}})^{\texttt{rd}} \\
 & & \quad \mathit{else\ if}\ (aa_e == \texttt{rb}) \\
 & & \quad \mathit{then}\ (\texttt{if}\ e^{\mathbf{aa}}\ s_1^{\mathbf{aa}}\ \texttt{else}\ s_2^{\mathbf{aa}})^{\texttt{rb}} \\
 & & \\
\mathcal{AC}_s\ (\texttt{while}\ e^{\mathbf{bt}}\ s^{\mathbf{bt}}) & ::= & \mathit{let}\ \ e^{\mathbf{aa}} = \mathcal{AC}_e\ e^{\mathbf{bt}} \\
 & & \quad aa_e = \mathit{outmost}\ \ e^{\mathbf{aa}} \\
 & & \quad s^{\mathbf{aa}} = \mathcal{AC}_s\ s^{\mathbf{bt}} \\
 & & \quad aa_s = \mathit{outmost}\ \ s^{\mathbf{aa}} \\
 & & \mathit{in}\ \ \mathit{if}\ (aa_e == \texttt{ev}) \wedge (aa_s == \texttt{ev}) \\
 & & \quad \mathit{then}\ (\texttt{while}\ e^{\mathbf{aa}}\ s^{\mathbf{aa}})^{\texttt{ev}} \\
 & & \quad \mathit{else\ if}\ (aa_e == \texttt{id}) \wedge (aa_s == \texttt{id}) \\
 & & \quad \mathit{then}\ (\texttt{while}\ e^{\mathbf{aa}}\ s^{\mathbf{aa}})^{\texttt{id}} \\
 & & \quad \mathit{else\ if}\ (aa_e == \texttt{ev}) \\
 & & \quad \mathit{then}\ (\texttt{while}\ e^{\mathbf{aa}}\ s^{\mathbf{aa}})^{\texttt{rd}} \\
 & & \quad \mathit{else\ if}\ (aa_e == \texttt{rb}) \\
 & & \quad \mathit{then}\ (\texttt{while}\ e^{\mathbf{aa}}\ s^{\mathbf{aa}})^{\texttt{rb}}
\end{array}
$$

Figure 2.8: Action analysis over a statement: Part 2

division about library parameters is used in preparing shared libraries for future specialization. Thus, we employ run-time specialization techniques in the framework of specialization of applications using shared libraries in order to deal with the intricacy associated with maintaining dynamic linking of specialized libraries.

Run-time partial evaluation typically performs BTA over the original program p to construct a binding-time annotated code $p^{\mathbf{bt}}$. A program generator generator *cogen* accepts $p^{\mathbf{bt}}$ and produces a program generator $p^{\mathbf{gen}}$ at compile-time, which is also termed **generating extension** in literature [35, 46]. $p^{\mathbf{gen}}$ creates the code $p_{in_1}^{\mathbf{fp}}$ at run-time when the concrete values of in_1 are available. We term this code produced

by the generating extension as its **footprint**. Formally,

Generating extension construction:	$p^{\mathbf{gen}} = [\![cogen]\!]\ p^{\mathbf{bt}}$
Footprint construction:	$p^{\mathbf{fp}}_{in_1} = [\![p^{\mathbf{gen}}]\!]\ in_1$
An equational definition of $p^{\mathbf{gen}}$:	$[\![p]\!]\ [in_1, in_2] = [\![\ [\![p^{\mathbf{gen}}]\!]\ in_1\ ]\!]\ in_2$

2.3.3 Structure of A Run-time Generating Extension

There are two notable partial evaluators supporting run-time specialization of the C language, namely Tempo [2, 25, 62, 63] and DyC [39, 38]. They both adopt a *template-based* approach to create generating extensions for libraries. A generating extension produced by these run-time specialization systems is commonly comprised of two parts: A **template file** and a **run-time specializer**. [2]

- A **template file** that encodes the dynamic expressions in the binding-time annotated code. It contains several program fragments each of which is (possibly) parameterized by a hole variable denoting the result of a static expression. Each program fragment is referred to as a **source template** and is delimited by symbolic labels to make sure the templates are considered in isolation by the compiler.

  When the template file is compiled into a binary, the information about the size and location of each compiled source template (which is termed as **object template** in Tempo) and the offset of each hole variable within the template are *also* extracted at compile-time to be used in constructing a run-time specializer.

- A **run-time specializer** that not only encodes the static expressions, but also contains operations to manipulate object templates. These operations include:

[2] The terms **template file** and **run-time specializer** are adopted from Tempo. DyC used the terms **template code** and **setup code** respectively

– Template dumping: This operation copies instructions of an object template into a dynamically allocated memory block and flushes the instruction cache to ensure its coherency.

  Every template captured by the template file is a candidate for dumping. The reason for dumping templates is that instantiated templates can be different from their original ones, because the former replace holes in the latter by values evaluated from static expressions.

– Hole filling: This operation writes the values evaluated from static expressions to the appropriate location of the hole variable in the dumped template. Hole filling is also termed **template instantiation** in the literature.

– Memory block allocating: A memory block is dynamically allocated at run-time to store all the templates that are selected, dumped and instantiated by the run-time specializer. The memory block forms the footprint of the generating extension.

The template file and run-time specializer are compiled and linked together to create a binary of the generating extension of a library.

Figure 2.9 presents the source code of a generating extension (i.e. the combination of a template file and a run-time specializer) of library `power` constructed by Tempo for the binding-time annotated code presented in Figure 2.4. For readability of presentation, we omit some unimportant details of these two files, e.g., in the run-time specializer the arguments of template dumping macro `DUMP_TEMPLATE` are simplified to the corresponding template identifier. Interested readers may wish to construct a run-time generation extension by themselves using Tempo to see the unsimplified source code.

The template file is parameterized by the dynamic parameter `b`. In this template file, there are four source templates `t0`, `t1`, `t2` and `t3`, as delimited by those symbolic

```
/** A template file **/

int N0;
int CH0;

int tmp_power (int b) {
  int z;
  t0_end:
  if (N0) {
    t1_start:
    return 1;
    t1_end:
  }
  else {
    t2_start:
    z=b*((int (*)(int))(&CH0))(b);
    return z;
    t2_end:
  }
  t3_start:
}

extern void _tmp_power () {
}
```

```
/** A run-time specializer **/

void *rts_power (int e) {
  char *spec_ptr;
  spec_ptr = get_code_mem(65536);
  DUMP_TEMPLATE(t0);
  if (e == 0)
    DUMP_TEMPLATE(t1);
  else {
    DUMP_TEMPLATE(t2);
    PATCH_CALL_HOLE(rts_power (e-1));
  }
  DUMP_TEMPLATE(t3);
  return (void *)spec_ptr;
}
```

Figure 2.9: A run-time generating extension of library `power` constructed by Tempo

labels. The static conditional test `e==0` is substituted by a dummy integer `N0`, whose role is to separate the two templates `t1` and `t2`. Either `t1` or `t2` is selected to be dumped by the run-time specializer based on the truth value of the static conditional test `e==0`. Template `t2` contains a static call hole variable `CH0` whose address is the one returned by each invocation of the (recursive) run-time specializer.

The run-time specializer is parameterized by the static parameter `e`. In the run-time specializer, the pointer `*spec_ptr` points to the beginning address of a memory block dynamically allocated by the instruction `get_code_mem` with a pre-fixed size 65536. The macros `DUMP_TEMPLATE` and `PATCH_HOLE` implement the dumping and instantiating template operations introduced above.

At run-time a footprint is created by executing the generating extension (more specifically, the run-time specializer) with respect to the values of static inputs. More specifically, a memory block is dynamically allocated to store all the templates that are selected, dumped and instantiated by the run-time specializer. Consider the generating extension of the `power` library depicted in Figure 2.9. Suppose the run-time specializer is called with the value of `e` as 2. The memory block dynamically allocated to form the footprint comprises the following sequence of object templates:

$$[\texttt{t0}, \texttt{t2}_1, \texttt{t0}, \texttt{t2}_0, \texttt{t0}, \texttt{t1}, \texttt{t3}, \texttt{t3}, \texttt{t3}]$$

where $\texttt{t2}_1$ and $\texttt{t2}_0$ are two object templates instantiated from original object template `t2` within which the static expression is filled with 1 and 0 respectively.

2.4 Our Framework for Specialization of Applications Using Shared Libraries

As mentioned in Section 1.3, there are three sub-problems of specialization of applications using shared libraries: independent specialization information generation, efficient specialized library construction and execution, and specialization engine enhancement. The following subsections provide an overview of our approach to address these problems.

2.4.1 Profitability Analysis

The perspective we adopt in independent specialization information generation is *how best to prepare a library for specialization such that the specialized libraries remains effective in as many applications as possible*. This perspective enables library developers to take advantage of the knowledge of the library implementation and prepare suitable specialization information for future deployment. Since a library implementation typically performs a case analysis over its deployment contexts, it inhibits effective specialization in the absence of information about its deployment contexts.

We develop a profitability analysis to automatically discover all effective binding-time divisions, which are termed as **binding-time signatures**, for a library without being aware of its deployment context.

We advocate using the term **profitability** to indicate the opportunity for specialization of a library, specifically *the ability to specialize conditional tests away at an earlier stage*. This is based on an effective heuristic that static reduction of conditional tests of `if` statements and static unrolling of `while` statements are the primary sources of profitable specialization both in terms of time and space. More specifically, this profitability can be divided into two categories: (1) **Direct profitability**: The ability to directly specialize away a conditional test inside a library; (2) **Indirect profitability**: The ability to specialize a library call so that the (direct or indirect) profitabilities inside the called library may be reaped.

Program Text	Profitability Point Annotations
`int power (int b, int e) {`	
`  int z ;`	
`  if (e == 0)`	/* profitability point 1*/
`    return 1 ;`	
`  else {`	
`    z = b * power(b, e − 1) ;`	/* profitability point 2*/
`    return z ;`	
`  }`	
`}`	

Figure 2.10: Library `power` annotated with profitability points information

Profitabilities residing inside a library can be identified by **profitability points**. Consider the library `power` given in Figure 2.10. There are a direct profitability and indirect profitability residing respectively at profitability points 1 and 2 in the library `power`, as illustrated in Figure 2.10.

When a library f is deployed in an application, we aim to attain **profitability fulfillment** which stands for the request that: (1) The binding time of one of the

conditional tests within the body of f is static, or; (2) The binding-time state established the library call site is deemed profitable with respect to the binding-time signatures of f.

In summary, *profitabilities are declared implicitly by identifying the profitability points inside the library, and the conceptual profitability declaration denotes the request to fulfill all or part of the (direct or indirect) profitabilities in the library.*

We have developed a modular profitability-oriented binding-time analysis whose main task is to convert the conceptual profitability declaration into a binding-time constraint, which is termed as **profitability signature**. A profitability signature of a library stipulates a binding-time condition enforced over the library's parameters in order to fulfill all or part of the profitabilities within a library.

For example, the profitability signature $\xi_{\texttt{power}}$ derived for library `power` is:

$$\xi_{\texttt{power}} \ ::= \ (bt_e == \texttt{s})$$

where bt_e is a binding-time variable pertaining to the library's parameter `e`. $\xi_{\texttt{power}}$ states that as long as the binding time of the parameter `e` is `s`, the profitability at points 1 and 2 can be fulfilled, regardless of the binding time of the parameter `b`. $\xi_{\texttt{power}}$ can also be expressed equivalently as a set of binding-time signatures of the library's parameters, as follows

$$\texttt{ss1} \ ::= \ (bt_b == \texttt{s}) \wedge (bt_e == \texttt{s})$$
$$\texttt{ss2} \ ::= \ (bt_b == \texttt{d}) \wedge (bt_e == \texttt{s})$$

2.4.2 Generic Specialization Component

Our vision adopted in specialization of applications using shared libraries is to *replace the original shared library with its generic specialization component (GSC for short) that caters for multiple specialization opportunities, while minimizing the need for*

repetitive and redundant specialization of libraries at the application level. Given that various binding-time divisions are produced when independently specializing a library through profitability analysis, a GSC inevitably accommodates different versions of the specialized libraries that are generated with respect to different binding-time signatures.

To achieve the objectives of efficient specialized library construction and execution we proposed in Section 1.3, it is important to manage and balance the trade-off between the multiplicity of specialized libraries and the space required for keeping them in order to exploit the sharing property.

GSC construction: The input to GSC construction is a set of action-annotated codes that are produced with respect to all the binding-time signatures returned by profitability analysis. The principle of constructing a GSC is to detect sharable templates by looking up each action annotated statement in the different action-annotated codes. Sharable templates are derived from identical action annotated statements. *All* distinct templates derived from different action annotated codes of library f are stored in a global **template repository** $f^{\mathbf{tmps}}$. We leverage the traditional two-part structure of a generating extension in constructing the GSC. A GSC $f^{\mathbf{gsc}}$ constructed for a library f is composed of a set of **local run-time specializers** $\{f^{\mathbf{rts}}_{ss_i}\}$ and a global **template repository** $f^{\mathbf{tmps}}$; the latter is shared by those local run-time specializers. Each local run-time specializer $f^{\mathbf{rts}}_{ss_i}$ is created from the corresponding action annotated code of a library with respect to a binding-time signature ss_i.

After a GSC $f^{\mathbf{gsc}}$ is constructed for a library f, it is ready for deployment in various applications. At compile-time $f^{\mathbf{gsc}}$ is instantiated with respect to a binding-time division ss established at application side. The instantiation returns a run-time generating extension $f^{\mathbf{ge}}_{ss}$. The $f^{\mathbf{ge}}_{ss}$ is composed of the corresponding $f^{\mathbf{rts}}_{ss}$ which contains

pointers to $f^{\mathbf{tmps}}$.

Footprint construction and execution: At run-time, a footprint $f^{\mathbf{fp}}_{val_s}$ is created from the generating extension $f^{\mathbf{ge}}_{ss}$ through executing $f^{\mathbf{rts}}_{ss}$ with respect to concrete values val_s for static input to f as specified in ss. $f^{\mathbf{fp}}_{val_s}$ is executed in a late stage with respect to concrete values val_d for the dynamic inputs to f specified in ss to produce the final output. The principle of constructing and executing a footprint is to *minimize the footprints of specialized shared libraries during execution.*

The templates stored in the template repository $f^{\mathbf{tmps}}$ can be divided into two categories. The first type of template does not contain any hole variables denoting results of static expressions and will remain unchanged during instantiation. The second type of template contains at least one hole variable to be instantiated by concrete values evaluated from static expressions at run-time. We term these two types of template as **totally dynamic templates** and **hybrid templates** respectively.

When creating a footprint at run-time from the generation extension $f^{\mathbf{ge}}_{ss}$, we maximize memory-sharing by choosing not to dump totally dynamic templates into the dynamically allocated memory block since they can be located in the memory block allocated for the global template repository. Only hybrid templates are dumped into a dynamically allocated memory block and instantiated by filling concrete values into their holes. Under this approach, the footprint is produced by linking the dumped hybrid templates in the dynamically allocated memory block and the totally dynamic templates found in the template repository.

As templates forming a footprint are not laid out in consecutive memory space, we need to connect them together so that execution of the footprint can proceed properly. We connect the templates in the following way:

1. The local run-time specializers build **address tables** when creating footprints.

An address table records a sequence of addresses of the object templates, depicting the program execution control flow among these templates during the execution of a footprint.

2. We add two types of operations for the purpose of passing program execution control among templates. These two operations capture the interactions between object templates and the address table.

   (a) The **registration operation** registers the address of an object template in the address table. In other words, it is considered as a static computation when concrete values of static inputs are available at run-time. Registration operations are part of a local run-time specializer.

   (b) The **redirecting operation** directs the program execution control to the subsequent template at the end of execution of current template whose address is recorded in the address table. In other words, it is considered as a dynamic computation when concrete values of dynamic inputs are available at run-time. Redirecting operations are inserted at the end of all templates, including both totally dynamic templates and instantiated hybrid templates.

Figure 2.11 gives an overview of the interactions between profitability analysis and GSC construction/execution described above.[3] The whole specialization process is composed of three essential elements: **Shared library specialization**, **application specialization** and **specialized application execution**

- **Shared library specialization**: This is a process that constructs a GSC for a shared library f by performing profitability analysis and GSC construction over f at compile-time. GSC materializes the profitability declaration and prepares f

[3]In a diagram, program or data values are in ovals, and processes are in boxes.

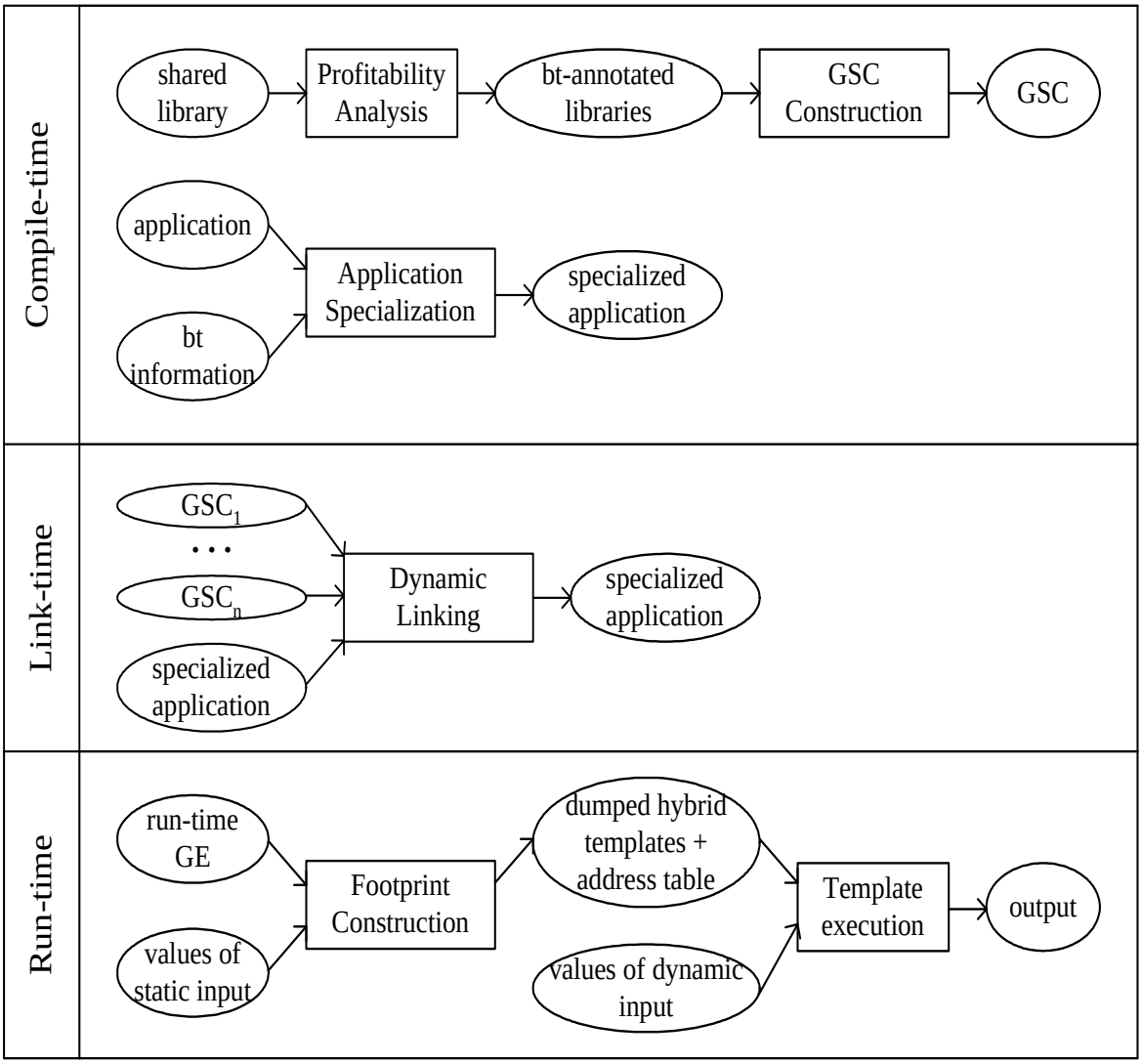

Figure 2.11: An overview of the interactions between profitability analysis and GSC construction/execution

for future specialization in various applications. It is an application-*independent* process.

- **Application specialization**: This is a process that installs the GSC with the applications and constructs a specialized application at compile-time with respect to a programmer-provided binding-time division of inputs by performing conventional BTA, action analysis and specialization over the application. Each action annotated library call is replaced by a call to the corresponding

GSC parameterized by its binding-time context: the GSC determines the most appropriate binding-time signature for this binding-time context, and returns a generating extension indexed by the selected binding-time signature.

Application programmers may also specify binding-time conditions for the called libraries at call sites inside the application as we proposed in [83]. If a relevant binding-time information is not provided, then the application becomes another shared library and thus we subject the application to the shared library specialization process.

- **Specialized application execution**: This is a process that runs the specialized application with respect to the concrete values for the whole input. The technique briefly described in this subsection regarding footprint construction/execution is employed to ensure the construction of minimal footprints throughout the execution.

  It is permissible for the application programmer to specify relevant binding-time information and the concrete values for the whole input all at once. For this case, we still maintain two separate phases: application specialization and specialized application execution.

2.4.3 Unification of Partial Evaluation and Program Slicing

We build a unified framework that theoretically captures the essence of both (static) program slicing and (offline) partial evaluation, and shows that these two techniques are intimately related.

This framework enables us to perceive both program slicing and partial evaluation as a three-stage process, namely: **residual analysis**, **action analysis** and **transformation**.

1. Residual analysis propagates specialization information throughout the program. In offline partial evaluation, BTA plays this role. Similarly, we define a

slicing analysis (either forward or backward) for this role in program slicing.

We claim that both BTA and slicing analysis are projection-based analyses [54] on well-classified information. Specifically, BTA is a projection-based analysis on static information, forward slicing analysis is a projection-based analysis on transient data, and backward slicing analysis is a projection-based analysis on residual data.

2. Action analysis uses information provided by residual analysis to determine the action to be taken at each program point.

   We associate static variables with transient variables, and dynamic variables with residual variables. It is satisfying to observe that *the decisions for removing/retaining a syntactic construct in program slicing are identical to the decisions for reducing/reconstructing a construct in partial evaluation*. That is, both program slicing and partial evaluation have identical action analysis, modulo the equivalence between static/dynamic and transient/residual.

3. The final stage, transformation, specializes a program based on the action decisions produced by the action analysis.

This unified framework enables us to assess both specialization techniques in a consistent manner, and to facilitate cross-fertilization between them.

CHAPTER 3

RELATED WORK

In this chapter, we review the literature published prior to and during the course of our research on the following three topics: Independent specialization information generation, management of specialized code, and unification of program slicing and partial evaluation. They correspond to the three sub-problems for specialization of applications using shared libraries, as introduced in Section 1.3.

3.1 Independent Specialization Information Generation

Schultz advocated the concept of *black box program specialization* in a position paper [69]. He proposed that library developers are responsible for identifying specialization opportunities of the library without taking into consideration the library's deployment contexts. He further proposed the release of specialization opportunities as an abstract specialization interface to application developers. He also advocated the kinds of features that a specialization framework should possess, such as automatic configuration based on the dependency information available in the library interface. However, Schultz only reports his initial investigations and some proposed solutions about this topic.

There have been several works that allow library developers to declare specialization information for libraries, free from their deployment contexts confined to any specific applications.

- In Tempo, library developers manually express their desired degree of optimization by declaring the expected specialization information for C function

definitions in a *specialization scenario* [24, 56, 57]. This specialization scenario is a binding-time information about *external specialization parameters*, which include functions, global variables and data structure intended to be specialized. A group of specialization scenarios for a C function definition are collected into one file named a *specialization module*.

A consistency check of the specialization scenarios is performed in a pre-phase analysis to ensure that the binding time information of external specialization parameters defined in the specialization scenario can be inferred at each reference point. More specifically,

  - if the check infers some data as dynamic but the library developer has declared it to be static, the partial evaluator will abort the specialization process with an error message signalling this mismatch, rather than producing an under-specialized program;
  - if the check infers some data as static but the library developer has declared it to be dynamic, the analysis considers the data to be dynamic, thus following the library developer's intentions rather than producing an over-specialized program.

- DyC [39, 38] is an annotation-directed partial evaluator for the C language based on a principled extension of partial evaluation. It allows library developers to express their specialization intentions inside program code by using a set of annotations, such as specialization primitives (e.g., `make_static`, `make_dynamic`, which specify the variables and code fragments on which dynamic compilation should take place) and specialization policy annotations (e.g., polyvariant or monovariant specialization, different caching policies, etc.). DyC does not create binding-time information explicitly as Tempo does. Instead, the set of internal annotations is compiled by DyC's specific compiler into information

guiding the construction of a run-time specializer for each dynamically compiled region. Most of the key cost/benefit trade-offs in the binding-time analysis and the run-time specialization are open to the library developer's control through declarative annotations.

- For object-oriented languages, Consel et al. introduced a *specialization class* as a language extension for Java language, aiming at expressing program specialization (not just specialization opportunities) in a separate and declarative way [6, 26, 79]. A specialization class specifies what methods should be specialized and what variables should be used for this specialization. Multiple specialization classes can be attached to a single regular class, capturing different opportunities for specialization. If these opportunities define a sequence of incremental specialization stages, the specialization classes can be extended step by step, instead of being all defined from scratch. That is, the specialization of a class is not fixed, but evolves as specialization values become available.

- Bobeff et al. [13, 14] proposed a brute force approach that systematically creates an exhaustive list of specialization information for a library, then allows library developers to intervene by manually removing those inconsistent specialization information or unsuitable specialization information, taking into consideration the benefits in terms of specialization opportunities. Unfortunately, this intervention makes the process of generating suitable specialization information an art only mastered by library developers with in-depth knowledge about partial evaluation.

3.2 Management of Specialized Code

There have been several works on managing specialized libraries that have been produced from a library with respect to the various specialization information before

deploying them in various applications.

- Bobeff et al. [13, 14] propose to collect all the specialized code created separately with respect to various specialization information into a *component generator*. This component generator is generated by a *component generator generator*, the input of which is a binding-time annotated code created with respect to a particular binding-time input information, and the output of which is a generating extension, called *service generator*. The latter generates a stream of strings representing the specialized code based on the specialization context. Each service generator is created for a binding-time input information. All service generators created with respect to various binding-time input information separately for a library are collected into a component generator for that library.

- Bhatia et al. [10] describe a *remote customization approach* to automatically generate highly optimized code that is then loaded and executed in the kernel of an embedded device. There are two key elements in the customization infrastructure:

  - *Context manager*: On the client side, the context manager extracts the customization values from the arguments of the application customization request. The context manager can also be configured to keep the number of specialized versions of a given module bounded. On the server side, the context manager processes the customization values in preparation for the customization phase. This task consists of storing these values in a customization table. The index in this table is a hash number corresponding to the original kernel memory address of the customization value.

  - *Code manager*: On the client side, the code manager maintains a cache of customized code indexed by the system call number and the customization context. This cache is shared across the application processes of the

device. The code manager runs in kernel mode and thus directly loads the customized code into the kernel, without using intermediate storage or buffering. Similar to the policies in the context manager, the code manager can be configured with a cache-replacement policy. On the server side, the code manager simply transmits the customized code to the device via the customized code channel.

- Schultz et al. [6, 70, 71, 72, 73] adopte an aspect-oriented approach in managing specialized code, which is an extension of the specialization class approach. This approach encapsulates the methods generated by a given specialization of the original object-oriented library into an aspect, and weaves the methods into the application during compilation. The technique of weaving is used to redirect existing function calls to calls to specialized functions. Access modifiers can be used to ensure that specialized methods can only be called from specialized methods encapsulated in the same aspect, and hence always be called from a safe context. Furthermore, the specialized libraries are cleanly separated from the original generic libraries, and can be removed from the library simply by deselecting the aspect.

These existing techniques managing for specialized libraries, do not take into consideration the issue of code duplication at either compile-time or run-time.

3.3 Unification of Program Slicing and Partial evaluation

Both program slicing and partial evaluation are well-developed specialization techniques, and are extensively discussed in the research community. To the best of our knowledge, the first work relating these two fields was done by Reps et al. [67]. They described a backward static slicing for strict functional programs through a

projection-based backward analysis. Later, Reps and his student described a partial evaluation using a dependence graph, thus placing both partial evaluation and program slicing on the same program representation [30]. Ochoa et al. [64] proposed a form of program specialization for lazy functional logic programs based on dynamic slicing. However, neither of them develop a unified framework to formally investigate the relation between these two techniques, and to produce new specialization technique that seamlessly combines the benefit of both partial evaluation and specialization.

In [77], Venkatesh proposed *quasi-static slicing*, the motivation of which arises from the situation that the values of some inputs are fixed while the behavior of the original program must be analyzed when the values of other inputs vary. Quasi-static slicing falls between static slicing and dynamic slicing, and is performed in *a similar spirit as partial evaluation*. However, this technique remains at the realm of program slicing, and fails to provide a uniform treatment on these two domains of specialization techniques.

Binkley et al. [11] explored similarities and differences between the specialized program of partial evaluation and a conditioned slice, [17, 19, 27, 28, 29, 42] and established a formal relationship between them. They used a program projection framework [40, 41] for the purpose of capturing the behavior of each transformation's algorithm sufficiently strongly to distinguish it from other algorithms. They claimed that the key semantic difference between program slicing and partial evaluation concerns the form of semantics preserved by each: program slicing preserves lazy semantics while partial evaluation preserves strict semantics. They also observed that the combination of partial evaluation transformations (i.e. reduce or residualize) together with syntax-preserving slicing could almost produce amorphous slicing [40, 41], in which the simplification is not limited to statement removal.

CHAPTER 4

PROFITABILITY ANALYSIS

The perspective we adopt in **independent specialization information generation** is *how best to prepare a library for specialization such that the specialized libraries remains effective in as many applications as possible*. This perspective enables library developers to take advantage of the knowledge of the library implementation and prepare suitable specialization contexts for future deployment. Since a library implementation typically performs a case analysis over its deployment context, the library inhibits effective specialization in the absence of these deployment contexts.

In this chapter, we study the technique of exploring specialization opportunities of a library and expressing these specialization opportunities via a programmer-friendly mechanism. In Section 4.1 we introduce the **profitability declaration** which is a novel methodology to capture specialization opportunities residing inside libraries independent of how the libraries are deployed. In Section 4.2 we introduce **profitability signature** which is a binding-time constraint stipulating a binding-time condition required for library parameters in order to fulfill the specialization opportunities conveyed by the profitability declaration. Then, in Section 4.3 we elaborate the **specialization policy** which sets the guidelines for computing profitability signatures and governs the specialization of a library in a specific application. In Section 4.4 we present a **profitability-oriented binding-time analysis** which generates a profitability signature for a library. We discuss termination aspect of partial evaluation in Section 4.5

To begin with, we highlight two terminologies that represent two categories of binding-time divisions under two different circumstances:

- **A binding-time signature**: It is a binding-time division of a library's parameters. It is derived at the library level and associated with the original library independent of the library's deployment contexts. This information serves as a guard to the specializations performed over the library.

- **A binding-time context**: It is a binding-time division of a library call's arguments. It is established at library call sites.

4.1 Profitability Declaration

There can be various kinds of specialization opportunities for a library. For instance, an application developer may wish to specialize a library handling generic array operations only when the latter is deployed in a context that deals with bit-vectors.

In this dissertation, we investigate a technique of turning a specialization opportunity of a library into a sufficient context that is amenable to partial evaluation. Specifically, we advocate the term **profitability** to indicate the specialization opportunity to *specialize conditional tests away in an earlier stage*. This is based on an effective heuristic that static reduction of conditional tests of `if` statements and static unrolling of `while` statements are primary sources of profitable specialization both in terms of time and space. An example is specializing away the conditional tests in network systems libraries [9]. Static conditional tests are also essential for making *function unfold* decisions when specializing.

This profitability can be divided into two categories:

1. **Direct profitability**: The ability to directly specialize away a conditional test inside a library;

2. **Indirect profitability**: The ability to specialize a library call so that the (direct or indirect) profitabilities inside the called library may be reaped.

Profitabilities residing inside a library are identified by **profitability points**. A

profitability point refers to a program segment that possesses a direct or indirect profitability. According to the classification of profitability described above, there are also two corresponding categories of profitability points: Conditional tests and library calls.

Consider a contrived example given in Figure 4.1. In this example, there is a direct profitability residing at profitability point 1 in the library `foo`, and a direct and indirect profitability residing respectively at profitability points 2 and 3 in the library `bar`, as highlighted in the comments.

Program Texts	Profitability Point Annotations
`int foo (int x, int y) {`	
`  if x > 0`	/* profitability point 1*/
`    return y+1;`	
`  else`	
`    return y-1;`	
`}`	
`int bar (int x, int y) {`	
`  if y > 0`	/* profitability point 2*/
`    return foo (x,y);`	/* profitability point 3*/
`  else`	
`    return 0;`	
`}`	

Figure 4.1: A contrived example demonstrating profitability point identification

When there are nested library calls within a conditional test or another library call, the nested library calls are identified as separate profitability points.

Consider a contrived example given in Figure 4.2. There are two profitability points residing in the library `g1`: One is an indirect profitability associated with the library call "`f1(x,y)`"; the other is a direct profitability for the whole conditional test "`f1(x,y) == 0`".

We term a library without any profitability point as a **plain library**.

Program Texts	Profitability Point Annotations

`int g1 (x, y) {`
`  if ( f1(x,y) == 0 )`
`  s1;`
`  else`
`  s2;`
`}`
	/* two profitability points identified */

Figure 4.2: A contrived example demonstrating nested profitability points

When a library f is deployed in an application, we aim to attain **profitability fulfillment** with the library f which stands for a binding-time request, such that:

1. The binding time of a conditional test in the library f is static, or;

2. The binding-time context established at a library call site in the library f is deemed profitable with respect to a binding-time signature of the called library.

In summary, *profitabilities are declared implicitly by identifying the profitability points inside the library, and the profitability declaration denotes the request to fulfill all or part of the (direct or indirect) profitabilities in the library*. Correspondingly, a specialization that fulfills all or part of the profitabilities available in a library is called a **profitable specialization**. Otherwise, it is termed an **unprofitable specialization**.

4.2 Profitability Signature

We convert the conceptual profitability declarations into a binding-time constraint. This binding-time constraint stipulates *a binding-time condition enforced over library parameters in order to fulfill all or part of the profitability in a library*. We term such a binding-time constraint as a **profitability signature** of a library. In line with the idea of design by contract [59], the profitability signature acts as a contract provided by a library.

4.2.1 Definition of A Binding-time Constraint

The syntax of a binding-time constraint is defined in Figure 4.3, where the definition of binding-time expression $\mathbf{BT_e}$ has been presented in Figure 2.3.

bt_e	$\in$	$\mathbf{BT_e}$	Binding-time expressions
ξ	$\in$	$\mathbf{BT_c}$	Binding-time constraints
	$::=$	$\texttt{true} \mid \texttt{false} \mid bt_{e_1} == bt_{e_2} \mid \xi_1 \wedge \xi_2 \mid \xi_1 \vee \xi_2$	

Figure 4.3: Syntax of binding-time constraint

There are three primitive binding-time constraints: Two binding-time constraint constants `true` and `false` which have the same meaning as the conventional truth values, and an equality constraint over two binding-time expressions bt_{e_1} and bt_{e_2}. A composite binding-time constraint is formed using two logical operators: logical conjunction $\wedge$ and logical disjunction $\vee$.

We denote the set of binding-time variables occurring in a binding-time constraint ξ by $Var(\xi)$. We denote the set of binding-time variables that are of interest in formulating the constraints (such as those pertaining to library parameters) by $\mathbf{Var}_{BT}$. $Var(\xi)$ is understood to be a subset of $\mathbf{Var}_{BT}$.

A **binding-time valuation** ϑ is an assignment of either `s` or `d` to all binding-time variables in $\mathbf{Var}_{BT}$. ϑ is of the type $\mathbf{Var}_{BT} \rightarrow \{\texttt{s}, \texttt{d}\}$. Given two binding-time valuations ϑ_1 and ϑ_2, $\vartheta_1 \sqsubseteq \vartheta_2$ *iff* $\forall bt_v \in \mathbf{Var}_{BT}, \vartheta_1(bt_v) \sqsubseteq \vartheta_2(bt_v)$

A **satisfiable binding-time valuation** ϑ of a binding-time constraint ξ is a binding-time valuation such that ξ is evaluated as `true`. We denote the set of satisfiable binding-time valuations of a binding-time constraint ξ by $Val(\xi)$.

The binding-time constraint constant `false` signifies that there is no satisfiable binding-time valuation, and the binding-time constraint constant `true` signifies that all 2^n combinations of binding-time valuations (where n is the cardinality of $\mathbf{Var}_{BT}$) are satisfiable.

4.2.2 An Example

For example, the profitability signatures derived for libraries `foo` and `bar` defined in Figure 4.1 are:

$$\xi_{\texttt{foo}} \ ::= \ (bt_x == \texttt{s})$$
$$\xi_{\texttt{bar}} \ ::= \ (bt_x == \texttt{s}) \vee (bt_y == \texttt{s})$$

$\xi_{\texttt{foo}}$ states that as long as the binding time of the parameter `x` is static, the profitability at profitability point 1 can be fulfilled, *regardless of the binding time of the parameter* `y`. $\xi_{\texttt{bar}}$ expresses a disjunctive condition in which the profitability at point 2 and point 3 can be fulfilled respectively: when the binding time of the parameter `y` is static, the profitability at profitability point 2 can be fulfilled; when the binding time of the parameter `x` is static, the indirect profitability at profitability point 3 can be fulfilled.

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

Note that a profitability signature of a library is not generated by a typical forward-fashion BTA with respect to a concrete binding-time division of the inputs. Instead, it is generated by propagating outwardly those *binding-time requests* at the profitability points. The algorithm for generating profitability signature is presented in Section 4.4.

4.3 Specialization Policy

Before detailing the algorithm of generating profitability signatures, we present the idea of a **specialization policy**, which describes how the profitability signature of a library is used when the library is deployed in various applications. The rationale for discussing the policy is that it provides the direction in which we design and discover profitability signatures.

4.3.1 Minimal Profitable Contexts

Specializations of a library when, the library is deployed in various applications, are governed by the binding-time contexts established at library call sites. A binding-time context is a binding-time constraint of the form:

$$\bigwedge\{bt_{v_i} == \mathtt{s} \mid \mathtt{d}\}$$

where $\{bt_{v_i}\}$ is a set of binding-time variables pertaining to the library parameters. The binding-time contexts are related to the profitability signature of the corresponding library through an **entailment** relation and **minimal profitable contexts** relation defined over binding-time constraints as follows.

Definition 4.1 (Entailment Relation)**.** *A binding-time constraint ξ_1 is said to entail another binding-time constraint ξ_2, denoted by $\xi_1 \vdash \xi_2$, iff for any satisfiable binding-time valuation $\vartheta_1 \in Val(\xi_1)$, there exists a satisfiable binding-time valuation $\vartheta_2 \in Val(\xi_2)$ such that $\vartheta_1 \sqsubseteq \vartheta_2$.*

The entailment relation is transitive, anti-symmetric and reflexive. As an example, a binding-time constraint $(bt_x == \mathtt{s}) \wedge (bt_y == \mathtt{s})$ entails another constraint $(bt_x == \mathtt{s}) \wedge (bt_y == \mathtt{d})$.

Since a binding-time valuation is an assignment of binding-time constants (`s` or `d`) to binding-time variables, it can also be expressed as a conjunction of equality constraints between binding-time variables and binding-time constants. This treatment enables us to establish entailment relations between binding-time valuations with other binding-time constraints. On the other hand, a binding-time context, which is of the form $\bigwedge\{bt_{v_i} == \mathtt{s} \mid \mathtt{d}\}$, can be treated as a binding-time valuation i.e., $\{bt_{v_i} \mapsto \mathtt{s} \mid \mathtt{d}\}$. These treatments are used in the next definition:

Definition 4.2 (Minimal Profitable Context)**.** *Let ϑ be a binding-time context (i.e., a binding-time valuation), ξ be a profitability signature (i.e., a binding-time constraint),*

and $\vartheta \vdash \xi$*. A minimal profitable context of* ϑ *with respect to* ξ *is a satisfiable binding-time valuation* ϑ_m *such that:*

$$\vartheta_m \in Val(\xi) \wedge \vartheta \sqsubseteq \vartheta_m \wedge$$
$$\forall \vartheta' \in Val(\xi) : (\vartheta_m \sqsubseteq \vartheta' \vee \ \neg(\vartheta_m \sqsubseteq \vartheta' \vee \vartheta' \sqsubseteq \vartheta_m))$$

Note that there may be *multiple* minimal profitable contexts of a binding-time context with respect to a profitability signature. For example, for a binding-time context $\vartheta ::= (bt_x == \mathtt{s}) \wedge (bt_y == \mathtt{s})$ and a profitability signature $\xi ::= ((bt_x == \mathtt{s}) \wedge (bt_y = \mathtt{d})) \vee ((bt_x = \mathtt{d}) \wedge (bt_y == \mathtt{s}))$, we have $\vartheta \vdash \xi$. The following ϑ_1 and ϑ_2 are both minimal profitable contexts of ϑ w.r.t. ξ.

$$\vartheta_1 \ ::= \ (bt_x == \mathtt{s}) \wedge (bt_y == \mathtt{d})$$
$$\vartheta_2 \ ::= \ (bt_x == \mathtt{d}) \wedge (bt_y == \mathtt{s})$$

Now, the specialization policy is defined as: Given a library f and its associated profitability signature ξ_f.

- If a binding-time context ξ_1 for a call to f entails ξ_f, then the library call will be specialized with respect to a minimal profitable context of ξ_1.
- Otherwise, all the binding times of f-call's arguments will be classified as dynamic. Thereafter, the binding time of f-calls returned value is dynamic.[1]

4.3.2 Two Examples in Applying a Specialization Policy

We now demonstrate how the specialization policy governs the specializations of libraries in applications by considering two examples.

[1]This is based on the assumption which has been introduced in Section 2.1 that: The return value of a library must (data- or/and control-) dependent on *all* the library's parameters

Example 1. Suppose library `foo` defined in Figure 4.1 is called in a specific application at two locations with different binding-time contexts $c_1 ::= (bt_x == \texttt{s} \wedge bt_y == \texttt{d})$ and $c_2 ::= (bt_x == \texttt{s} \wedge bt_y == \texttt{s})$ respectively, where bt_x and bt_y are binding-time variables pertaining to respectively the first and second parameters of library `foo`. According to Definition 4.2, c_1 and c_2 are both minimal profitable contexts of $\xi_{\texttt{foo}}$. Thus each of the two calls will be specialized with respect to the exact binding-time context correspondingly.

We do not collapse the specializations of these two calls into one specialized code, even though the two contexts only differ in their binding-times for parameter `y`. Thus, this policy allows us to *explore static information as much as possible in the presence of profitable specialization*.

Example 2. Consider two inter-related libraries `add` and `mul` defined in Figure 4.4. They implement the arithmetic addition and multiplication operations respectively. They are labeled for convenient reference in Subsection 4.4.3.

Library `add` is a plain library and its profitability signature is `false`. The two calls to `add` in the library `mul` can only be specialized with all their arguments as dynamic regardless of the binding times that `x`, `y` and `z` hold before the calls. After the call, the binding times of the arguments of `add` will be restored to those values before the call.

This treatment of temporarily classifying the binding time of an argument to dynamic is similar to the *raise operation* used in many existing partial evaluators, such as [15]. We would like to highlight that the raise operation over the arguments of library calls is only performed at the library side when the library accepts a binding-time context. The interaction between libraries and applications will be explained in detail in Chapter 5.

Program Points	Program Texts
	`int add(int m, int n) {`
1 :	`  return m + n;`
	`}`
	`int mul(int x, int y) {`
1 :	`  int z;`
2 :	`  if (x > 0) {`
3 :	`  z = mul(x-1,y);`
4 :	`  return add(z, y);`
	`  }`
5 :	`  if (x == 0)`
6 :	`  return 0;`
7 :	`  if (y > 0) {`
8 :	`  z = mul(x,y-1);`
9 :	`  return add(z,x);`
	`  }`
10 :	`  return mul(-x, -y);`
	`}`

Figure 4.4: Libraries `add` and `mul`

4.4 Profitability-oriented Binding-time Analysis

We have developed and implemented a modular **profitability-oriented BTA** (abbreviated as *PA*) to perform profitability analysis and compute a profitability signature of a library. The specifications of *PA* are defined in Figures 4.5, 4.6, 4.7 and 4.8. Before describing the analysis in detail, we highlight two essential features that distinguish our *PA* from a conventional BTA, and then introduce the two primary data structures used in the analysis.

- **Feature 1 of *PA*:** Our *PA* does not perform a conventional forward-fashion BTA with respect to a concrete binding-time division of the inputs. Instead, the analysis propagates binding-time requests at profitability points outwardly and generates the profitability signature of the library. All binding-time information generated by our *PA* is parameterized by the binding-time variables pertaining

to the library's parameters.

- **Feature 2 of *PA*:** The specialization policy also requires our *PA* to take into consideration the condition for the raise operation (which is termed **raise condition**) that: the condition fails when a library call is specialized with respect to the minimal profitable context of a binding-time context; the condition succeeds when a library call is specialized with respect to a configuration that all the binding times of library call's arguments are dynamic.

- **Two primary data structures used in the analysis:**

  - The first primary data structure is a binding-time environment $\rho \in \mathbf{BT_{env}}$, which is associated with each program point. ρ is a set of mappings of the type $\mathbf{Var} \rightarrow \langle \mathbf{bte} : \mathbf{BT_e}, \mathbf{raise} : \mathbf{BT_c} \rangle$ where

    * **Var** ranges over program variables occurring at the program point;
    * The field **bte** records a binding-time expression of the variable, which is computed solely based on the data- and control-dependencies;
    * The field **raise** records a raise condition, which is expressed as a binding-time constraint. Algorithmically, a raise condition fails (succeeds) when the binding-time constraint is evaluated to `true` (`false`).

    Overall, the binding time of a variable is decided by not only its associated binding-time expression *bte* but also its associated raise condition *raise*: It is static iff the *bte* is evaluated to `s` *and raise* is evaluated to `true` with respect to a concrete binding-time division of inputs; otherwise, it is dynamic.

    Again, as highlighted in the **feature 1 of our *PA***, all binding-time information stored in ρ is parameterized by the binding-time variables pertaining to the library's parameters.

– The second data structure is a global library-name-indexed table $\tau \in \mathbf{F_{tab}}$, which is needed to deal with inter-library analysis. τ is a set of mappings of the type $\mathbf{FName} \rightarrow \langle \mathbf{ps} : \mathcal{P}(\mathbf{Var}), \quad \mathbf{pro} : \mathbf{BT_c} \rangle$, where

  * **FName** ranges over the library's names;
  * The field **ps** records the library's parameters;
  * The field **pro** records a binding-time constraint to form the library's profitability signature.

4.4.1 Specification of the Analysis

$$
\begin{array}{l}
PA_{fs} \ \in \ (\mathbf{FDef})^{\mathbf{n}} \rightarrow (\mathbf{BT_c})^{\mathbf{n}} \\
\\
PA_{fs} \ fd_1, \ldots, fd_n ::= \\
\ let \ \ \tau_0 = InitFTab \ \ fd_1, \ldots, fd_n \\
\qquad for \ \ (i = 1; i++; i \leq n) \ \ \{ \\
\qquad\quad fn_i = fd_i.\mathbf{name}; \\
\qquad\quad s_i = fd_i.\mathbf{body}; \\
\qquad\quad ps_i = fd_i.\mathbf{paras}; \\
\qquad\quad \rho_i = InitBTEnv \ ps_i; \\
\qquad\quad \langle \_ , \ \tau_i \rangle \ = \ PA_s \ s_i \ \rho_i \ \tau_{i-1} \ fn_i \ \mathtt{s} \ \ \} \\
\qquad \mathcal{F} \ \langle btcv_1, \ \ldots, \ btcv_n \rangle = \langle \tau_n(fn_1).\mathbf{pro}, \ \ldots, \ \tau_n(fn_n).\mathbf{pro} \rangle \\
\ in \ \ \bigsqcup_{n \geq 0} \mathcal{F}^n \ \langle \mathtt{false}, \ \ldots, \ \mathtt{false} \rangle \\
\\
InitFTab \ \ fd_1, \ldots, fd_n ::= \{ fd_i.\mathbf{name} \mapsto \langle \mathbf{ps} : fd_i.\mathbf{paras}, \ \mathbf{pro} : \mathtt{false} \rangle \mid 1 \leq i \leq n \} \\
\\
InitBTEnv \ \ ps ::= \{ v_i \mapsto \langle \mathbf{bte} : bt_{v_i} \ , \ \mathbf{raise} : \mathtt{true} \rangle \mid v_i \in ps \}
\end{array}
$$

Figure 4.5: Profitability-oriented BTA over (inter-related) libraries

Description of PA_{fs} (defined in Figure 4.5): The main specification PA_{fs} takes in a set of (interrelated) libraries fd_1, ..., fd_n, and returns the profitability signatures for each library. $fd.\mathbf{name}$, $fd.\mathbf{paras}$ and $fd.\mathbf{body}$ retrieve respectively the name, parameters and body (i.e., a sequence of statements) of a library fd_i.

PA_{fs} initializes a global table τ_0 in the way that at each entry of τ_0 the field **pro** is assigned with `false`. Then, it performs profitability analysis PA_s over each library

body, updates the corresponding entry in the global table τ with the binding-time constraint denoting the library's profitability signature.

After performing profitability analysis over the body of the last library fd_n, the binding-time constraints stored in the **pro** field of each entry of τ_n are retrieved to form a function $\mathcal{F}$. $\mathcal{F}$ takes in a tuple of **binding-time constraint variables** $btcv_1, \ldots, btcv_n$, each of which refers to the corresponding binding-time constraint stored in $\tau_n(fn_i)$.**pro**. $\mathcal{F}$ returns a tuple of the binding-time constraints stored in the **pro** field of each entry of τ_n, each of which may contain binding-time constraint variables since libraries may be mutually recursive. The introduction of the binding-time constraint variables into a binding-time constraint denoting the library's profitability signature will be explained later in the description of PA_e

Finally, PA_{fs} performs a least fixed point computation over the function $\mathcal{F}$ to obtain a set of binding-time constraints without binding-time constraint variables. Each such binding-time constraint is the ultimate profitability signature for a library. The least fixed point computation starts from the bottom value `false`, which represents the profitability signature of a plain function. We will show an example of this least fixed point computation in Section 4.4.3.

Description of PA_s (defined in Figures 4.6 and 4.7): PA_s takes in a statement s, a binding-time environment ρ at the entry of s, a global table τ, the name fn of the library in which s resides, a binding-time expression ctr of a conditional test for the purpose of maintaining control-dependency information, and returns the updated ρ and τ at the exit of s.

$$PA_s \in \mathbf{Stat} \to \mathbf{BT_{env}} \to \mathbf{F_{tab}} \to \mathbf{FName} \to \mathbf{BT_e} \to \langle \mathbf{BT_{env}},\ \mathbf{F_{tab}} \rangle$$

$PA_s\ (v = e)\ \rho\ \tau\ fn\ ctr ::=$
$\quad let \quad (bt_e,\ raise_e,\ pro_e) = PA_e\ e\ \rho\ \tau$
$\qquad\quad bt_v = bt_e \sqcup ctr$
$\qquad\quad \rho' = \rho[v \leftarrow \langle \mathbf{bte} : bt_v\ ,\ \mathbf{raise} : raise_e \rangle]$
$\qquad\quad pro_{final} = \tau(fn).\mathbf{pro} \vee pro_e$
$\qquad\quad \tau' = \tau[fn \leftarrow \langle \mathbf{ps} : \tau(fn).\mathbf{ps},\ \mathbf{pro} : pro_{final} \rangle]$
$\quad in \quad \langle \rho',\ \tau' \rangle$

$PA_s\ (\texttt{int}\ v_1,\ \ldots,\ \texttt{int}\ v_n)\ \rho\ \tau\ fn\ ctr ::=$
$\quad let \quad \{bt_{v_i} = newBTVar(v_i) \mid 1 \leq i \leq n\}$
$\qquad\quad \rho' = \rho \bowtie_\rho \bigcup_{1 \leq i \leq n} \{v_i \mapsto \langle \mathbf{bte} : \mathtt{d}\ ,\ \mathbf{raise} : \mathtt{true} \rangle\}$
$\quad in \quad \langle \rho',\ \tau \rangle$

$PA_s\ (\texttt{return}\ e)\ \rho\ \tau\ fn\ ctr ::=$
$\quad let \quad \langle bt_e,\ raise_e,\ pro_e \rangle = PA_e\ e\ \rho\ \tau$
$\qquad\quad ret$ be a fresh variable name
$\qquad\quad bt_{ret} = bt_e \sqcup ctr$
$\qquad\quad \rho' = \rho \bowtie_\rho \{ret \mapsto \langle \mathbf{bte} : bt_{ret}\ ,\ \mathbf{raise} : raise_e \rangle\}$
$\qquad\quad pro_{final} = \tau(fn).\mathbf{pro} \vee pro_e$
$\qquad\quad \tau' = \tau[fn \leftarrow \langle \mathbf{ps} : \tau(fn).\mathbf{ps}\ ,\ \mathbf{pro} : pro_{final} \rangle]$
$\quad in \quad \langle \rho',\ \tau' \rangle$

Figure 4.6: Profitability-oriented BTA over a statement : Part 1

The operators $\bowtie_\rho$, $\uplus_\rho$ and $\uplus_\tau$ used in PA_s are defined as:

- $\bowtie_\rho$ extends a binding-time environment (its left operand) with new entries (its right operand).

- $\rho_1 \uplus_\rho \rho_2 ::= \{x \mapsto \langle \mathbf{bte} : bte_1 \sqcup bte_2\ ,\ \mathbf{raise} : raise_1 \wedge raise_2 \rangle \mid x \mapsto \langle \mathbf{bte} : bte_1\ ,\ \mathbf{raise} : raise_1 \rangle \in \rho_1\ ,\ x \mapsto \langle \mathbf{bte} : bte_2\ ,\ \mathbf{raise} : raise_2 \rangle \in \rho_2\}$.

  As mentioned in the introduction of the data structures used in the analysis (at the beginning of Section 4.4), the binding-time constraints stored in the field **raise** of a binding-time environment ρ are for the purpose of recording raise conditions. At the exit of an `if` statement, the binding time of a variable x is static only when the raise conditions in two branches of the `if` statement

PA_s (`if` e s_1 `else` s_2) ρ τ fn ctr ::=
 let $\langle bt_e,\ raise_e,\ pro_e\rangle = PA_e\ e\ \rho\ \tau$
 $ctr' = ctr \sqcup bt_e$
 $\langle \rho_1,\ \tau_1\rangle = PA_s\ s_1\ \rho\ \tau\ fn\ ctr'$
 $\langle \rho_2,\ \tau_2\rangle = PA_s\ s_2\ \rho\ \tau\ fn\ ctr'$
 $\rho' = \rho_1 \uplus_\rho \rho_2$
 $\tau_{merg} = \tau_1 \uplus_\tau \tau_2$
 $pro_{if} = (bt_e == \mathtt{s}) \wedge raise_e$ /** **profitability fulfillment condition**
 $pro_{final} = \tau_{merg}(fn).\mathbf{pro} \vee pro_{if} \vee pro_e$
 $\tau' = \tau_{merg}[fn \leftarrow \langle \mathbf{ps} : \tau_{merg}(fn).\mathbf{ps}\ ,\ \mathbf{pro} : pro_{final}\rangle]$
 in $\langle \rho', \tau'\rangle$

PA_s (`while` e s) ρ τ fn ctr ::=
 let $\langle bt_e,\ raise_e,\ pro_e\rangle = PA_e\ e\ \rho\ \tau$
 $ctr' = ctr \sqcup bt_e$
 $\langle \rho',\ \tau'\rangle = PA_s\ s\ \rho\ \tau\ fn\ ctr'$
 $pro_{while} = (bt_e == \mathtt{s}) \wedge raise_e$ /** **profitability fulfillment condition**
 $pro_{final} = (\tau'(fn).\mathbf{pro} \vee pro_{while} \vee pro_e)$
 in $if\ \rho = \rho'$
 $then$ let $\tau_{final} = \tau'[fn \leftarrow \langle \mathbf{ps} : \tau'(fn).\mathbf{ps}\ ,\ \mathbf{pro} : pro_{final}\rangle]$
 in $\langle \rho,\ \tau_{final}\rangle$
 $else$ PA_s (`while` e s) ρ' τ' fn ctr'

PA_s (s_1;s_2) ρ τ fn ctr ::=
 let $\langle \rho_1,\ \tau_1\rangle = PA_s\ s_1\ \rho\ \tau\ fn\ ctr$
 in $PA_s\ s_2\ \rho_1\ \tau_1\ fn\ ctr$

Figure 4.7: Profitability-oriented BTA over a statement : Part 2

both fails (i.e., the corresponding binding-time constraints are both evaluated to `true`). So we calculate the conjunction of the two binding-time constraints $raise_1$ and $raise_2$ of the same variable x.

- $\tau_1 \uplus_\tau \tau_2 ::= \{fn \leftarrow \langle \mathbf{ps} : ps\ ,\ \mathbf{pro} : pro_1 \vee pro_2\rangle \mid fn \mapsto \langle \mathbf{ps} : ps\ ,\ \mathbf{pro} : pro_1\rangle \in \tau_1,\ fn \mapsto \langle \mathbf{ps} : ps\ ,\ \mathbf{pro} : pro_2\rangle \in \tau_2\}$.

  As mentioned in the introduction of the global library-name-indexed table τ, the binding-time constraints stored in the field **pro** of τ are for the purpose of

recording a profitability signature. So we perform $\vee$ over two binding-time constraints pro_1 and pro_2 to express a disjunctive condition in which profitabilities at two profitability points are fulfilled.

$$
\begin{array}{l}
PA_e \ \in \ \mathbf{Exp} \rightarrow \mathbf{BT_{env}} \rightarrow \mathbf{F_{tab}} \rightarrow \langle \mathbf{BT_e},\ \mathbf{BT_c},\ \mathbf{BT_c} \rangle \\
\\
PA_e\ c\ \rho\ \tau ::= \langle \mathtt{s},\ \mathtt{true},\ \mathtt{false} \rangle \\
\\
PA_e\ v\ \rho\ \tau ::= \langle \rho(v).\mathbf{bte},\ \rho(v).\mathbf{raise},\ \mathtt{false} \rangle \\
\\
PA_e\ (e_1\ b_{op}\ e_2)\ \rho\ \tau ::= \\
\ let \quad \langle bt_{e_1},\ raise_{e_1},\ pro_{e_1} \rangle = PA_e\ e_1\ \rho\ \tau \\
\qquad \langle bt_{e_2},\ raise_{e_2},\ pro_{e_2} \rangle = PA_e\ e_2\ \rho\ \tau \\
\ in \quad \langle bt_{e_1} \sqcup bt_{e_2},\ raise_{e_1} \wedge raise_{e_2},\ pro_{e_1} \vee pro_{e_2} \rangle \\
\\
PA_e\ fn\ (e_1, \ldots, e_n)\ \rho\ \tau ::= \\
\ let \quad \{\langle bt_{e_i},\ raise_{e_i}\ pro_{e_i} \rangle = PA_e\ e_i\ \rho\ \tau \mid 1 \leq i \leq n\} \\
\qquad bt_{fn} = bt_{e_1} \sqcup \ldots \sqcup bt_{e_n} \\
\qquad \{v_1, \ldots, v_n\} = \tau(fn).\mathbf{ps} \\
\qquad pro = btcv_{fn} \wedge (bt_{v_1} == bt_{e_1}) \ldots \wedge (bt_{v_n} == bt_{e_n}) \\
\qquad pro_{fn} = pro \wedge raise_{e_1} \wedge \ldots \wedge raise_{e_n} \ /^{**}\ \mathbf{profitability\ fulfillment\ condition} \\
\qquad pro_{final} = pro_{fn} \vee pro_{e_1} \vee \ldots \vee pro_{e_n} \\
\ in \quad \langle bt_{fn},\ pro_{fn},\ pro_{final} \rangle
\end{array}
$$

Figure 4.8: Profitability-oriented BTA over an expression

Description of PA_e (defined in Figure 4.8): PA_e takes in an expression e, a binding-time environment ρ at the entry of a statement s where e occurs, a global table τ, and returns the binding-time expression of e which is computed solely based on data- and control-dependencies, a binding-time constraint recording its raise condition, and a binding-time constraint standing for a profitability fulfillment condition produced at the exit of e.

Profitability fulfillment condition generation: PA_s and PA_e play central roles in generating binding-time constraints, at each profitability point (i.e., a conditional test or a library call) to establish a condition of profitability fulfillment as we have

explained in Section 4.1. The binding-time constraints are expressed in terms of the binding-time expressions of the variables appearing at those program points. Specifically:

- For a conditional test e of an `if` or `while` statement: We request the binding time of the conditional test e to be static. The profitability fulfillment condition thus generated is $(bt_e == \texttt{s}) \wedge raise_e$ where bt_e is the binding-time expression of e and $raise_e$ is the raise condition associated with this e.

- For a library call expression $fn(e_1, \ldots, e_n)$: We request the binding-time contexts established at the library call site to be deemed *profitable* with respect to the profitability signature of library fn. The profitability fulfillment condition thus generated is $btcv_{fn} \wedge (bt_{v_1} == bt_{e_1}) \wedge \ldots \wedge (bt_{v_n} = bt_{e_n}) \wedge raise_{e_1} \wedge \ldots \wedge raise_{e_n}$ where

  - $btcv_{fn}$ is a binding-time constraint variable whose value is the binding-time constraint stored in $\tau(fn)$.**pro** which is expressed in terms of $\{bt_{v_i} \mid 1 \leq i \leq n\}$;
  - $\{bt_{v_i} \mid 1 \leq i \leq n\}$ are binding-time variables pertaining to library fn's parameters $\{v_i \mid 1 \leq i \leq n\}$ respectively;
  - $\{bt_{e_i} \mid 1 \leq i \leq n\}$ are the binding-time expressions of fn-call's arguments respectively; and
  - $\{raise_{e_i} \mid 1 \leq i \leq n\}$ are raise conditions generated for fn-call's arguments $\{e_i \mid 1 \leq i \leq n\}$ respectively.

All the binding-time constraints generated at the profitability points of a library fn are collected in a form of disjunctive binding-time constraint and stored into $\tau(fn)$.**pro** at the exit of the library fn. This disjunctive binding-time constraint

expresses a disjunctive condition in which the profitability at various profitability points can be fulfilled respectively.

4.4.2 Soundness of Profitability-oriented Binding-time Analysis

The soundness of profitability-oriented binding-time analysis can be expressed in terms of **profitable specialization** which is defined as follows:

Definition 4.3 (Profitable Specialization)**.** *Specialization of a call to library f with respect to a binding-time context ξ is said to be a profitable specialization if either of the following conditions holds during specialization:*

1. *One of library f's direct profitability points will have the binding-time expression of its conditional test evaluated to static, or*

2. *One of the library calls within the body of f will be specialized with respect to a context ξ' that will result in a profitable specialization.*

Theorem 4.1 (Soundness)**.** *Given a program $P = \{fd_1, \ldots, fd_n\}$, let $PA_{fs}(P) = \tau$. For any i between 1 and n, let fn_i be the name of the library fd_i. For any binding-time context ξ of a library call to fd_i such that $\xi \vdash \tau(fn_i).\mathbf{pro}$, specialization of the library fd_i with respect to ξ will be profitable.*

Sketch of Proof The proof of soundness of profitable specialization is formulated as a proof of the profitability-oriented BTA which is defined in Figures 4.6, 4.7 and 4.8. More specifically, our claim is that:

> $\forall\ \xi \vdash \tau(fn_i).\mathbf{pro} \Rightarrow \exists\ btc$, which is a profitability fulfilment conditions generated by *PA* for the library fd_i, such that *btc* is evaluated to `true` with respect to ξ

The sketch of the proof is as follows:

1. For the case that a library fd_i does not contain any library call:

   As described in Figure 4.7, the profitability fulfilment conditions generated at the direct profitability points (i.e., the conditional tests of `if` or `while` statements) are of the following form, where e is a conditional test expression

   $$(bt_e == \mathtt{s}) \wedge raise_e$$

   These profitability fulfilment conditions can all be simplified to

   $$(bt_e == \mathtt{s})$$

   because $raise_e$ is always `true` in this case. Thereafter, $\tau(fn_i).\mathbf{pro}$ is literally identical as a disjunction of those simplified profitability fulfilment conditions, which can be written as

   $$\tau(fn_i).\mathbf{pro} \equiv (bt_{e_1} == \mathtt{s}) \vee \ldots \vee (bt_{e_n} == \mathtt{s})$$

   $\xi \vdash \tau(fn_i).\mathbf{pro}$

   $\Rightarrow$ ξ is a satisfiable valuation of the binding-time constraint $\tau(fn_i).\mathbf{pro}$

   $\Rightarrow$ $\exists i \in \{1, \ldots, n\} : (bt_{e_i} == \mathtt{s})$ is evaluated to `true` with respect to ξ

2. For the case that a library fd_i contains library calls:

   Without loss of generality, we assume there is only one call to a library fd_j in the body of the library fd_i. As described in Figure 4.8, the profitability fulfilment condition generated for a library call to fd_j is of the following form, where $btcv_{fn_j}$ refer to the corresponding binding-time constraint stored in $\tau_n.(fn_j).\mathbf{pro}$; $v_1, \ldots, v_n$ are parameters of the library fd_j; and $e_1, \ldots, e_n$ are arguments of a call to the library fd_j:

   $$btcv_{fn_j} \wedge (bt_{v_1} == bt_{e_1}) \ldots \wedge (bt_{v_n} == bt_{e_n}) \wedge raise_{e_1} \wedge \ldots \wedge raise_{e_n}$$

   Each of the profitability fulfilment conditions can be rewritten as a conjunctive normal form $btcv_{fn_j} \wedge simpler\_btc$ by using the distributive and the absorption laws, where $simpler\_btc$ is a binding-time constraint without binding-time

constraint variables. A least fixed point computation is performed over the function:

$$\mathcal{F}\ \langle btcv_1,\ \ldots,\ btcv_n\rangle = \langle \tau(fn_1)\mathbf{pro},\ \ldots,\ \tau(fn_n).\mathbf{pro}\rangle$$

to resolve these binding-time constraint variables. This least fixed point computation is formulated as follows[2]:

- Step 1: Let: $btcv_1^1 = \tau^1(fn_1).\mathbf{pro} = \texttt{false}$; ..., $btcv_n^1 = \tau^1(fn_n).\mathbf{pro} = \texttt{false}$, then $\mathcal{F}$ is updated to:

$$\mathcal{F}\langle btcv_1, \ldots, btcv_n\rangle = \langle \tau^2(fn_1).\mathbf{pro}, \ldots, \tau^2(fn_n).\mathbf{pro}\rangle$$

- Step m: Let $btcv_1^m = \tau^m(fn_1).\mathbf{pro}; \ldots, btcv_n^m = \tau^m(fn_n).\mathbf{pro}$, then $\mathcal{F}$ is updated to:

$$\mathcal{F}\langle btcv_1, \ldots, btcv_n\rangle = \langle \tau^{m+1}(fn_1).\mathbf{pro}, \ldots, \tau^{m+1}(fn_n).\mathbf{pro}\rangle$$

The following proof by induction is conducted over least fixed point computation steps. The τ and *btcv* shown in the statement of the proof are the updated result at the end of each step of the least fixed point computation.

(a) **Base case:** The binding-time constraint variables are initially assigned the bottom value `false`, which represents the unprofitable specialization of a library. Then, we have

$$\xi \vdash \tau^1(fn_i).\mathbf{pro} \equiv \xi \vdash btcv_{fn_i} \equiv \xi \vdash \texttt{false} \equiv \texttt{false}$$

Our claim thus becomes *vacuously true*.

(b) **Inductive hypothesis:**

[2]a superscription Dat^n associated with a data structure *Dat* stands for the value of that *Dat* computed at n-th step

- $\forall\, \xi_i \vdash \tau^{n-1}(fn_i).\mathbf{pro} \Rightarrow$ there exists a profitability fulfillment condition $btcv^{n-1}_{fn_j} \wedge simpler\_btc$ that is evaluated to true with respect to ξ_i
- $\forall\ \xi_j \vdash \tau^{n-1}(fn_j).\mathbf{pro} \Rightarrow$ there exists a profitability fulfillment condition that is evaluated to true with respect to ξ_j

(c) **Inductive case:** Given $\xi_i \vdash \tau^n(fn_i).\mathbf{pro}$, we need to check if the profitability fulfillment condition $btcv^n_{fn_j} \wedge simpler\_btc$ is evaluated to true with respect to ξ_i. There are three sub-cases in the inductive case according to how libraries fd_i and fd_j are interacted.

i. Sub-case 1: The library fd_j is a *leaf* library, i.e., there are no library calls within fd_j: There is no binding-time constraint variable in the $\tau(fn_j).\mathbf{pro}$. Thereafter, the profitability fulfillment condition associated with a call to fn_j does not contain binding-time constraint variable. Then similar to what we have discussed in the case "that a library fd_i does not contain any library call", $\tau(fn_i).\mathbf{pro}$ is literally identical to a disjunction of its profitability fulfillment conditions. The proof is similar with the previous one.

ii. Sub-case 2: fd_j and fd_i are identical, i.e., a library call to fd_j within the library fd_i is a self-recursive function call: Then we need to check if the profitability fulfillment condition $btcv_{fn_i} \wedge simpler\_btc$ is evaluated to true with respect to ξ_i.

$$
\begin{aligned}
& btcv^n_{fn_i} \equiv \tau^n(fn_j).\mathbf{pro} \\
\equiv\ & (\tau^{n-1}(fn_i).\mathbf{pro} \wedge simpler\_btc) \wedge simpler_b tc \\
\equiv\ & \tau^{n-1}(fn_i).\mathbf{pro} \wedge simpler\_btc
\end{aligned}
$$

By induction hypothesis, $\tau^{n-1}(fn_i).\mathbf{pro} \wedge simpler\_btc$ is evaluated to true with respect to ξ_i. This proves the sub-case 2.

iii. Sub-case 3: fd_j and fd_i are mutually recursive:

$$btcv^{n}_{fn_j} \equiv \tau^{n}(fn_j).\mathbf{pro} \equiv \tau^{n-1}(fn_j).\mathbf{pro} \wedge simpler\_btc$$

$$\equiv \quad btcv^{n-1}_{fn_j} \wedge simpler\_btc$$

$$btcv^{n}_{fn_j} \wedge simpler\_btc \equiv (btcv^{n-1}_{fn_j} \wedge simpler\_btc) \wedge simpler\_btc$$

$$\equiv \quad btcv^{n-1}_{fn_j} \wedge simpler\_btc$$

By induction hypothesis, $btcv_fnj \wedge simpler\_btc$ is evaluated to true with respect to ξ_i. This proves the sub-case 3.

This proves the result. □

4.4.3 An Example

Pp	**Binding-time environments of the library** `add`
1	$\{\mathtt{m} \mapsto \langle \mathtt{bt_m}, \mathtt{true}\rangle,\ \mathtt{n} \mapsto \langle \mathtt{bt_n}, \mathtt{true}\rangle,\ \mathtt{ret} \mapsto \langle \mathtt{bt_m} \sqcup \mathtt{bt_n}, \mathtt{true}\rangle\}$
Pp	**Binding-time environments of the library** `mul`
1	$\{\mathtt{z} \mapsto \langle \mathtt{bt_z},\ \mathtt{true}\rangle\}$
2	$\{\mathtt{x} \mapsto \langle \mathtt{bt_x},\ \mathtt{true}\rangle\}$
3	$\{\mathtt{x} \mapsto \langle \mathtt{bt_x},\ \mathtt{true}\rangle, \mathtt{y} \mapsto \langle \mathtt{bt_y},\ \mathtt{true}\rangle, \mathtt{z} \mapsto \langle \mathtt{bt_x} \sqcup \mathtt{bt_y},\ \xi_1\rangle\}$
4	$\{\mathtt{y} \mapsto \langle \mathtt{bt_y},\ \mathtt{true}\rangle, \mathtt{z} \mapsto \langle \mathtt{bt_x} \sqcup \mathtt{bt_y},\ \xi_1\rangle, \mathtt{ret} \mapsto \langle \mathtt{bt_x} \sqcup \mathtt{bt_y},\ \xi_1 \wedge \xi_2\rangle\}$
5	$\{\mathtt{x} \mapsto \langle \mathtt{bt_x},\ \mathtt{true}\rangle\}$
6	$\{\mathtt{ret} \mapsto \langle \mathtt{bt_x},\ \mathtt{true}\rangle\}$
7	$\{\mathtt{y} \mapsto \langle \mathtt{bt_y},\ \mathtt{true}\rangle\}$
8	$\{\mathtt{x} \mapsto \langle \mathtt{bt_x},\ \mathtt{true}\rangle, \mathtt{y} \mapsto \langle \mathtt{bt_y}, \mathtt{true}\rangle, \mathtt{z} \mapsto \langle \mathtt{bt_x} \sqcup \mathtt{bt_y},\ \xi_1\rangle\}$
9	$\{\mathtt{x} \mapsto \langle \mathtt{bt_x},\ \mathtt{true}\rangle, \mathtt{z} \mapsto \langle \mathtt{bt_x} \sqcup \mathtt{bt_y},\ \xi_1\rangle, \mathtt{ret} \mapsto \langle \mathtt{bt_x} \sqcup \mathtt{bt_y},\ \xi_1 \wedge \xi_2\rangle\}$
10	$\{\mathtt{x} \mapsto \langle \mathtt{bt_x},\ \mathtt{true}\rangle, \mathtt{y} \mapsto \langle \mathtt{bt_y}, \mathtt{true}\rangle, \mathtt{ret} \mapsto \langle \mathtt{bt_x} \sqcup \mathtt{bt_y},\ \xi_1\rangle\}$
	where
	$\xi_1 ::= \mathtt{btcv_{mul}} \wedge (\mathtt{bt_x} == \mathtt{bt_x}) \wedge (\mathtt{bt_y} == \mathtt{bt_y})$
	$\xi_2 ::= \mathtt{btcv_{add}} \wedge (\mathtt{bt_m} == \mathtt{bt_x} \sqcup \mathtt{bt_y}) \wedge (\mathtt{bt_n} == \mathtt{bt_y})$

Table 4.1: Binding-time environments generated for libraries `mul` and `add`

Tables 4.1 and 4.2 depict the binding-time environments and profitability fulfillment conditions generated for the libraries `mul` and `add` before conducting the least fixed-point computation to compute the profitability signatures. The column **Pp** contains the statement labels of the libraries `mul` and `add`.

Pp	Profitability fulfillment conditions of the library `mul`
2	$\mathtt{btc}_1 ::= \mathtt{bt}_\mathtt{x} == \mathtt{s}$
3	$\mathtt{btc}_2 ::= \mathtt{btcv}_\mathtt{mul} \wedge (\mathtt{bt}_\mathtt{x} == \mathtt{bt}_\mathtt{x}) \wedge (\mathtt{bt}_\mathtt{y} == \mathtt{bt}_\mathtt{y})$
4	$\mathtt{btc}_3 ::= \mathtt{btcv}_\mathtt{add} \wedge (\mathtt{bt}_\mathtt{x} \sqcup \mathtt{bt}_\mathtt{y} == \mathtt{bt}_\mathtt{m}) \wedge (\mathtt{bt}_\mathtt{y} == \mathtt{bt}_\mathtt{n}) \wedge \xi_1$
5	$\mathtt{btc}_4 ::= \mathtt{bt}_\mathtt{x} == \mathtt{s}$
7	$\mathtt{btc}_5 ::= \mathtt{bt}_\mathtt{y} == \mathtt{s}$
8	$\mathtt{btc}_6 ::= \mathtt{btcv}_\mathtt{mul} \wedge (\mathtt{bt}_\mathtt{x} == \mathtt{bt}_\mathtt{x}) \wedge (\mathtt{bt}_\mathtt{y} == \mathtt{bt}_\mathtt{y})$
9	$\mathtt{btc}_7 ::= \mathtt{btcv}_\mathtt{add} \wedge (\mathtt{bt}_\mathtt{x} \sqcup \mathtt{bt}_\mathtt{y} == \mathtt{bt}_\mathtt{m}) \wedge (\mathtt{bt}_\mathtt{y} == \mathtt{bt}_\mathtt{n}) \wedge \xi_1$
10	$\mathtt{btc}_8 ::= \mathtt{btcv}_\mathtt{mul} \wedge \mathtt{bt}_\mathtt{x} == \mathtt{bt}_\mathtt{x} \wedge \mathtt{bt}_\mathtt{y} == \mathtt{bt}_\mathtt{y}$

Table 4.2: Profitability fulfillment conditions generated for libraries `mul`

The global table τ generated before performing the least-fixed point computation is:

$$\tau ::= \{\mathtt{add} \mapsto \langle\{\mathtt{m},\mathtt{n}\}, \mathtt{false}\rangle, \quad \mathtt{mul} \mapsto \langle\{\mathtt{x},\mathtt{y}\}, \mathtt{btc}_1 \vee \ldots \vee \mathtt{btc}_8\rangle\}$$

Binding-time constraints btc_3 and btc_7 can be simplified to `false` based on the knowledge that $\mathtt{btcv}_\mathtt{add} = \tau(\mathtt{add}).\mathbf{pro} = \mathtt{false}$.

The other binding-time constraints can be simplified using the distributive and the absorption laws. Correspondingly, the binding-time constraint $\mathtt{btc}_1 \vee \ldots \vee \mathtt{btc}_8$ which denotes the profitability signature of `mul` is simplified to:

$$\mathtt{btcv}_\mathtt{mul} \vee (\mathtt{bt}_\mathtt{x} == \mathtt{s}) \vee (\mathtt{bt}_\mathtt{y} == \mathtt{s})$$

The function $\mathcal{F}$ thus constructed is

$$\mathcal{F}\ \langle \mathtt{btcv}_\mathtt{add},\ \mathtt{btcv}_\mathtt{mul}\rangle = \langle \mathtt{false},\ \mathtt{btcv}_\mathtt{mul} \vee (\mathtt{bt}_\mathtt{x} == \mathtt{s}) \vee (\mathtt{bt}_\mathtt{y} == \mathtt{s})\rangle$$

Finally, we conduct least fixed-point operation on $\mathcal{F}$ to obtain the ultimate profitability signature of library `mul` without the binding-time constraint variables $\mathtt{btcv}_\mathtt{add}$, $\mathtt{btcv}_\mathtt{mul}$:

$$(\mathtt{bt}_\mathtt{x} == \mathtt{s}) \vee (\mathtt{bt}_\mathtt{y} == \mathtt{s})$$

The binding-time variables included in the set of binding-time environments, which are produced by the profitability-oriented BTA, will be *instantiated* with respect to the satisfiable binding-time valuations of the profitability signature. These

instantiated binding-time information facilitate construction of the GSC which will be described in the next chapter.

4.4.4 Binding-time Signatures in Practice

As required by the specialization policy, all the binding times of a library call's arguments are raised to dynamic if a minimal profitable context cannot be found for a binding-time context for that call. Correspondingly, to prepare a library for all possible cases (i.e., profitable and unprofitable) of specialization, we include the *all-dynamic* binding-time signatures in the complete set of binding-time signature attached with a library.

Table 4.3 at the end of this chapter lists the complete set of binding-time signatures associated with other sample libraries. The profitable binding-time signatures are underlined.

4.5 Termination Aspect of Partial Evaluation

Our development of profitability analysis is orthogonal to the issue of termination of partial evaluation. In practice, termination of partial evaluation is usually controlled by adjusting the binding-time signature of libraries. Certainly, existing techniques in ensuring termination of partial evaluation (such as [7, 37]) can be added to our analysis. Alternatively, annotations can be included in the library body to ensure that the binding-time signature generated does not lead to non-termination of the partial evaluation process. One such technique is to introduce an **assert** annotation which asserts that the binding times of some (usually non-inductive) parameters of a library should be more dynamic than other (inductive) parameters [83]. Consider a contrived example given in Figure 4.9

When `mc` is specialized with respect to a specialization context "$(\texttt{bt}_\texttt{x} == \texttt{s}) \wedge (\texttt{bt}_\texttt{y} == \texttt{d})$", the second recursive call is placed under a dynamic control and the binding-times of the two arguments of the second recursive call are inferred as `s`

Program Points	Program Text
1	`int mc (int x, int y) {`
2	`  if x > 0`
3	`    return mc(x-1,y-1);`
4	`  else if y > 0`
5	`    return mc(x-1,y-1);`
6	`  else return y;`
	`}`

Figure 4.9: A contrived example used to demonstrate the usage of **assert** annotations

and `d` respectively. Infinite specializations occur under this specialization context. A snapshot of an infinite specialization, when the value of `x` is 1, is shown in Figure 4.10.

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

Figure 4.10: A snapshot of an infinite specialization

We introduce the notion of **assert** to curb such infinite specialization from arising. An **assert** annotation declares a binding-time constraint over the variables available at a particular program point. The syntax of the assert ξ is defined in Figure 4.11.

bt_e	$\in$	$\mathbf{BT_e}$	Binding-time expressions
ξ	$\in$	**Assert**	Assert annotation
	$::=$	`true` $\mid$ `false` $\mid$ $bt_{e_1} == bt_{e_2} \mid bt_{e_1} <= bt_{e_2} \mid$ $\xi_1 \wedge \xi_2 \mid \xi_1 \vee \xi_2$	

Figure 4.11: Syntax of **assert** annotations

For the `mc` library defined in Figure 4.9, we can provide an **assert** annotation "$\texttt{bt}_\texttt{x} == \texttt{bt}_\texttt{y}$" at the function header. This **assert** annotation demands that the binding-times of both parameters of the `mc` library should always be the same when

specializing the mc library. For a specialization to take place, the related assertion must be met. The use of **assert** annotations reflects our desire to control specialization effectiveness in a declarative fashion. In effective, these asserts capture the decision of poor man's generalization on library parameters, as described in [16, 43].

A more refined use of **assert** is also possible: placing different **assert** annotations at different library calls occurring within a library. For the mc example, we can provide two **assert** annotations at program points 2 and 4, respectively:

$$2: \quad \mathtt{bt_x} <= \mathtt{bt_y}$$

$$4: \quad \mathtt{bt_y} <= \mathtt{bt_x}$$

The above **assert** annotations 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 "$(\mathtt{bt_x} == \mathtt{s}) \wedge (\mathtt{bt_y} == \mathtt{d})$", we will allow the first call to be specialized aggressively, but the second call will not be specialized. The specialized code thus produced is presented in Figure 4.9.

```
int mc_1(int y) {
  return mc_0 (y-1);
}
```

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

Figure 4.12: Specialized code of the library mc

Note that the specialization policy described in Section 4.3 is a policy generally applied in independent library specialization, no matter how the profitability signature is generated for a library. In other words, the specialization policy also governs the usage of such *refined* profitability signature generated by adding those extra binding-time constraints.

4.6 Summary

Instead of manually declaring specialization opportunities as summarized in Section 3.1, our profitability signatures are derived automatically through identifying profitability points within a library. In this work, we choose to identify conditional tests occurring in `if` and `while` statements as a source for profitable specialization. This technique resembles that proposed by Mock [60] for his profiling tool Calpa. Calpa automatically generates declarations for the specializer Dyc for C programs. As Calpa handles C applications rather than libraries, it is able to perform dynamic analysis over an application to discover specialization opportunities. Doing so for a library could be more difficult, since analyzing an application that deploys a library will only uncover part of the specialization opportunities of the library.

Our specialization policy, which only allows specialization when the specialization context entails 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 [31, 32]. We believe the current definition of profitability, which guarantees elimination of conditional tests in `if` and `while` statements, is acceptable as being practical by the partial evaluation community.

We have implemented a prototype tool deriving profitability signatures of shared libraries. The tool contains 12 files written in the OCaml language [3], and it has 1752 lines of code in total. This tool and analysis results of some testcases are publicly available at the web site

`http://www.comp.nus.edu.sg/~zhuping/prototype/pa`

Program texts	Binding-time signatures

int power (int b, int e) {
  int z;
  if (e==0)
  return 1;
  else {
  z = b * power(b, e-1);
  return z;
  }
}
	ss1 ::= (bt_b == s) ∧ (bt_e == s)
	ss2 ::= (bt_b == d) ∧ (bt_e == s)
	ss3 ::= (bt_b == d) ∧ (bt_e == d)

int power_loop (int b, int e) {
  int z;
  z = 1;
  while (e > 0) {
  z = b * z;
  e = e - 1;
  }
  return(accum);
}
	loop_ss1 ::= (bt_b == s) ∧ (bt_e == s)
	loop_ss2 ::= (bt_b == d) ∧ (bt_e == s)
	loop_ss3 ::= (bt_b == d) ∧ (bt_e == d)

int ack (int m, int n) {
  if (m==0)
  return n+1;
  else if (n==0)
  return ack(m-1, 1);
  else {
  int tmp;
  tmp = ack(m, n-1);
  return ack(m-1, tmp);
  }
}
	ack_ss1 ::= (bt_b == s) ∧ (bt_e == s)
	ack_ss2 ::= (bt_b == d) ∧ (bt_e == s)
	ack_ss3 ::= (bt_b == s) ∧ (bt_e == d)
	ack_ss4 ::= (bt_b == d) ∧ (bt_e == d)

int gcd (int m, int n) {
  if (m<n) {
  int tmp;
  tmp = m;
  m = n;
  n = tmp;
  }
  if (n ==0)
  return m;
  else
  return gcd(n, m % n);
}
	gcd_ss1 ::= (bt_m == s) ∧ (bt_n == s)
	gcd_ss2 ::= (bt_m == d) ∧ (bt_n == d)

Table 4.3: The complete set of binding-time signatures derived for other sample libraries

CHAPTER 5

GENERIC SPECIALIZATION COMPONENT

The vision adopted in **efficient specialized library construction and execution** is to replace original libraries with their generic specialization components (GSC for short) that cater for various specialization opportunities, while minimizing the need for repetitive and redundant specialization of libraries at application level. Given that multiple binding-time signatures may be produced when independently specializing a library through **profitability analysis** as described in Chapter 4, the generic specialization component typically accommodates different versions of the specialized libraries which are generated with respect to different binding-time signatures.

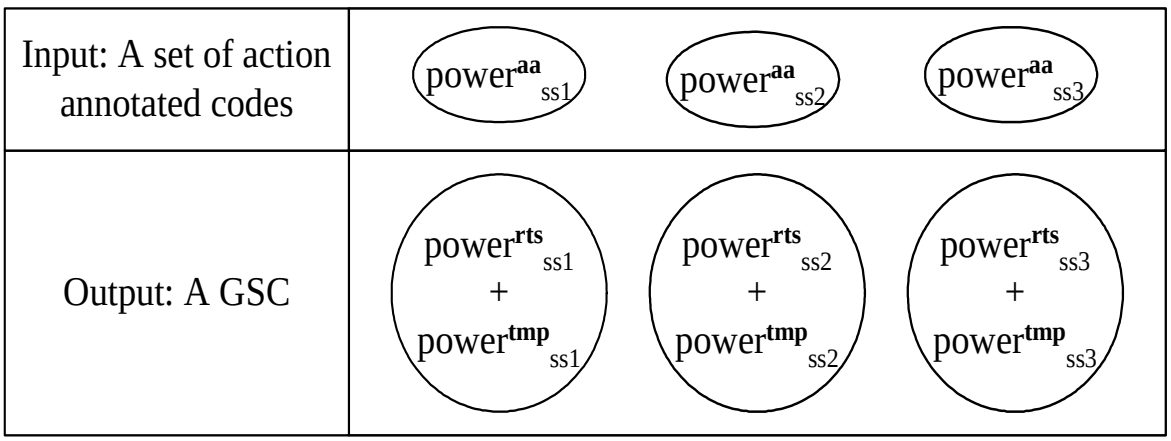

Figure 5.1: Traditional approach to construct a GSC for library `power` with respect to three binding-time signatures

The most straightforward way to construct a GSC, as has been done by most of the traditional approaches that have been surveyed in Section 3.2, is to create a set of specialized libraries with respect to the various binding-time signatures. Figure 5.1 illustrates an example of a GSC constructed according to the traditional approach for library `power` with respect to three binding-time signatures, where `ss1`, `ss2` and

ss3 were defined in Table 4.3.

The template files $\texttt{power}_{\texttt{ss2}}^{\textbf{tmp}}$ and $\texttt{power}_{\texttt{ss3}}^{\textbf{tmp}}$ are presented in Figure 5.2.[1] Readers are suggested to refer back to Section 2.3.3 for the introduction of template files produced by Tempo. Note that the two different template files share several identical templates, as underlined in the figure.

```
/** Template file power_ss2^tmp **/

int N2, CH2;
int tmp_power_2 (int b){
  int z;
  t0_end:
  if (N2) {
    t1_start:
    return 1;
    t1_end:}
  else {
    t2_start:
    z = b * ((int (*)(int))(&CH2))(b);
    return z;
    t2_end:}
  t3_start:
}
```

```
/** Template file power_ss3^tmp **/

int CH3;
int tmp_power_3 (int b, int e){
  int z;
  if (e == 0)
    return 1;
  else {
    z = b * ((int(*)(int, int))(&CH3))(b, e - 1);
    return z;}
}
```

Figure 5.2: Two template files adapted from Tempo

[1]The template file $\texttt{power}_{\texttt{ss1}}^{\textbf{tmp}}$ is empty since there is no dynamic expression in the binding-time annotated code generated with respect to the binding-time signature ss1.

To achieve the objectives of efficient specialized library construction and execution we set in Section 1.3, it is important to manage and balance the trade-off between the multiplicity of specialized libraries and the space required for keeping them in order to exploit the sharing property. As these specializable libraries are generated beforehand at compile-time, they enable maximal sharing during compile-time before being deployed in multiple applications. In addition, in this chapter we propose to enable sharing *both* at compile-time when constructing a GSC for a library *and* at run-time when executing the GSC inside applications.

The rest of this chapter is organized as follows. Section 5.1 and 5.2 elaborate the principles we adopt to enable sharing when constructing and executing a GSC at compile-time and run-time respectively through illustrative examples. The algorithms for GSC construction and execution are detailed in Section 5.3. In Section 5.4 we describe our experimental studies of GSC approach. We summarize our approach and survey related work in reducing duplication in partial evaluation in Section 5.5.

For ease and readability of presentation, we define some notational conventions used in this chapter and explain them as follows:

- $f^{\mathbf{gsc}}$: The GSC of a library f
- $f^{\mathbf{aa}}_{ss}$: Action annotated code of the library f constructed with respect to a binding-time signature ss
- $f^{\mathbf{ge}}_{ss}$: A generating extension of the library f constructed with respect to a binding-time signature ss
- $f^{\mathbf{rts}}_{ss}$: A run-time specializer of the library f constructed with respect to a binding-time signature ss
- $f^{\mathbf{fp}}_{val_s}$: The footprint of the library f constructed by executing its corresponding generating extension with respect to the values of static variables val_s

5.1 Principle of GSC Construction

The input to GSC construction is a set of action-annotated codes that are the result of performing conventional action-analysis over the set of binding-time annotated codes returned by profitability analysis. The principle of constructing a GSC is to *detect sharable templates by looking up each action-annotated statement in the different action-annotated codes*. Sharable templates are derived from identical action-annotated statements.

5.1.1 Template Repository Construction

We build a **global template repository** $f^{\mathbf{tmps}}$ which captures all distinct templates derived from different action annotated codes of f. Table 5.1 lists all the distinct templates derived from the three action annotated codes of the library `power` depicted in Figure 5.3.

Label	Template
gt0	`int z;`
gt1	`return 1 ;`
gt2	`z = b * ((int (*)(int))(&CH2))(b);`
gt3	`return z;`
gt4	`if (e == 0)`
gt5	`z = b * ((int(*)(int, int))(&CH3))(b, e - 1);`

Table 5.1: Distinct templates derived from the three action-annotated codes of library `power`

The algorithm to derive the distinct templates and the layout of the global template repository file will be presented in Section 5.3.

5.1.2 Two-part Structure of GSC

We leverage the traditional two-part structure of a generating extension in constructing a GSC. A GSC $f^{\mathbf{gsc}}$ is composed of a set of **local run-time specializers** and a

<table>
<tr>
<td>

$/**\ \mathtt{power}^{\mathbf{aa}}_{\mathtt{ss1}}\ **/$

$\mathtt{int\ power\ (int\ b^{ev},\ int\ e^{ev})\ \{}$
$\quad \mathtt{(int\ z)^{ev}\ ;}$
$\quad \mathtt{if^{ev}\ (e == 0)^{ev}}$
$\quad\quad \mathtt{(return\ 1)^{ev}\ ;}$
$\quad \mathtt{else\ \{}$
$\quad\quad \mathtt{(z = b * power(b,\ e - 1))^{ev}\ ;}$
$\quad\quad \mathtt{(return\ z)^{ev}\ ;}$
$\quad \}$
$\}$

</td>
<td>

$/**\ \mathtt{power}^{\mathbf{aa}}_{\mathtt{ss2}}\ **/$

$\mathtt{int\ power\ (int\ b^{id},\ int\ e^{ev})\ \{}$
$\quad \mathtt{(int\ z)^{id}\ ;}$
$\quad \mathtt{if^{rd}\ (e == 0)^{ev}}$
$\quad\quad \mathtt{(return\ 1)^{id}\ ;}$
$\quad \mathtt{else\ \{}$
$\quad\quad \mathtt{(z = b^{id} * power(b^{id},\ e^{ev} - 1)^{rb})^{rb}\ ;}$
$\quad\quad \mathtt{(return\ z)^{id}\ ;}$
$\quad \}$
$\}$

</td>
</tr>
<tr>
<td>

$/**\ \mathtt{power}^{\mathbf{aa}}_{\mathtt{ss3}}\ **/$

$\mathtt{int\ power\ (int\ b^{id},\ int\ e^{id})\ \{}$
$\quad \mathtt{(int\ z)^{id}\ ;}$
$\quad \mathtt{if^{id}\ (e == 0)^{id}}$
$\quad\quad \mathtt{(return\ 1)^{id};}$
$\quad \mathtt{else\ \{}$
$\quad\quad \mathtt{(z = b * power(b,\ e - 1))^{id};}$
$\quad\quad \mathtt{(return\ z)^{id};}$
$\quad \}$
$\}$

</td>
<td>

$/**\ \mathtt{Binding-timesignatures}\ **/$

$\mathtt{ss1}\ ::\ (\mathtt{bt_b} == \mathtt{s}) \wedge (\mathtt{bt_e} == \mathtt{s})$
$\mathtt{ss2}\ ::\ (\mathtt{bt_b} == \mathtt{d}) \wedge (\mathtt{bt_e} == \mathtt{s})$
$\mathtt{ss3}\ ::\ (\mathtt{bt_b} == \mathtt{d}) \wedge (\mathtt{bt_e} == \mathtt{d})$

</td>
</tr>
</table>

Figure 5.3: Action-annotated code constructed for the library `power` with respect to three binding-time signatures

global template repository; the latter being shared by all local run-time specializers. Each local run-time specializer pertains to the specialization of the library with respect to a distinct binding-time signature.

Figure 5.4 depicts an illustrative example of a GSC constructed by our approach for the library `power` with respect to three binding-time signatures: Each of those three local run-time specializers $\mathtt{power}^{\mathbf{rts'}}_{\mathtt{ss1}}$, $\mathtt{power}^{\mathbf{rts'}}_{\mathtt{ss2}}$ and $\mathtt{power}^{\mathbf{rts'}}_{\mathtt{ss3}}$ is responsible for performing the static computations and manipulating templates stored in the sharable template repository. The run-time specializers constructed in our approach are different from those produced by traditional run-time specialization techniques, as illustrated by Figure 5.1, since we adopt different run-time specialization mechanisms

in manipulating templates, which will be explained in Section 5.2 and presented in more detail in Section 5.3.

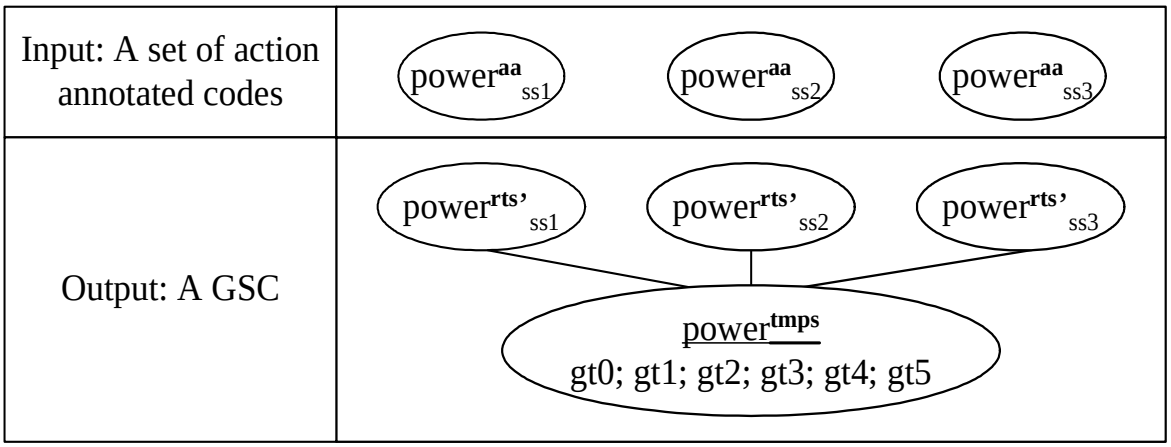

Figure 5.4: Illustration of constructing GSC for library `power` in our approach

It is worth reminding the reader that a GSC is constructed to handle *all* uses of the specialized shared library, rather than the uses relevant to a specific application. Although this implies that a GSC can contain several local run-time specializers for many different binding-time signatures, the size of the GSC is curbed by the fact that:

1. The number of such binding-time signatures is limited to the *profitable* ones, as explained in Chapter 4, and

2. Templates are shared in the global template repository.

Our GSC construction not only maximizes sharing at compilation-time, it also paves the way for maximizing memory-sharing at run-time since the GSC exists in shared library form and it is amenable to *memory-sharing* at run-time.

After a GSC $f^{\mathbf{gsc}}$ is constructed for a library f with respect to a set of binding-time signatures, it is ready for deployment in various applications. It is instantiated with respect to a specialization context ss established at the application side before

run-time and produces a generating extension $f_{ss}^{\mathbf{ge}}$ indexed by ss. $f_{ss}^{\mathbf{ge}}$ is composed of the corresponding $f_{ss}^{\mathbf{rts}}$ which refers to the templates in $f^{\mathbf{tmps}}$.

5.2 Principle of Footprint Construction and Execution

At run-time, a footprint $f_{val_s}^{\mathbf{fp}}$ is created from the generating extension $f_{ss}^{\mathbf{ge}}$ through executing $f_{ss}^{\mathbf{rts}}$ with respect to concrete values val_s for the static inputs to f as specified in ss. $f_{val_s}^{\mathbf{fp}}$ is executed in the late stage with respect to concrete values val_d for dynamic inputs to f specified in ss to produce the final output. The principle of constructing and executing a footprint is to *minimize the footprints of specialized shared libraries during execution*.

5.2.1 Methodology for Dumping Fewer Templates

The templates stored in the template repository can be divided into two categories. The first type of template is not embedded with any hole variables denoting results of static expressions and will remain unchanged during instantiation. The second type of template contains at least one hole variable to be instantiated by concrete values evaluated from static expressions at run-time. We term these two types of template **totally dynamic templates** and **hybrid templates** respectively.

When creating a footprint at run-time from the generation extension returned by $f^{\mathbf{gsc}}(ss)$, we maximize memory-sharing by choosing not to dump totally dynamic templates into the dynamically allocated memory block since they can be located in the memory block allocated for the global template repository. Only hybrid templates are dumped into a dynamically allocated memory block and instantiated by filling concrete values into their holes. Operationally, dumping of hybrid templates is performed by dumping operations instrumented in the local run-time specializers of a GSC. On the other hand, totally dynamic templates need not be associated with dumping operations.

Figure 5.5 shows the layouts of memory blocks dynamically allocated for storing the footprints of `power` generated with respect to inputs 2 and 3, respectively, by our approach and by a traditional approach. $\texttt{gt2}_2$, $\texttt{gt2}_1$ and $\texttt{gt2}_0$ are three templates instantiated from the original hybrid template `gt2` within which the static holes are filled with 2, 1 and 0 respectively.[2]

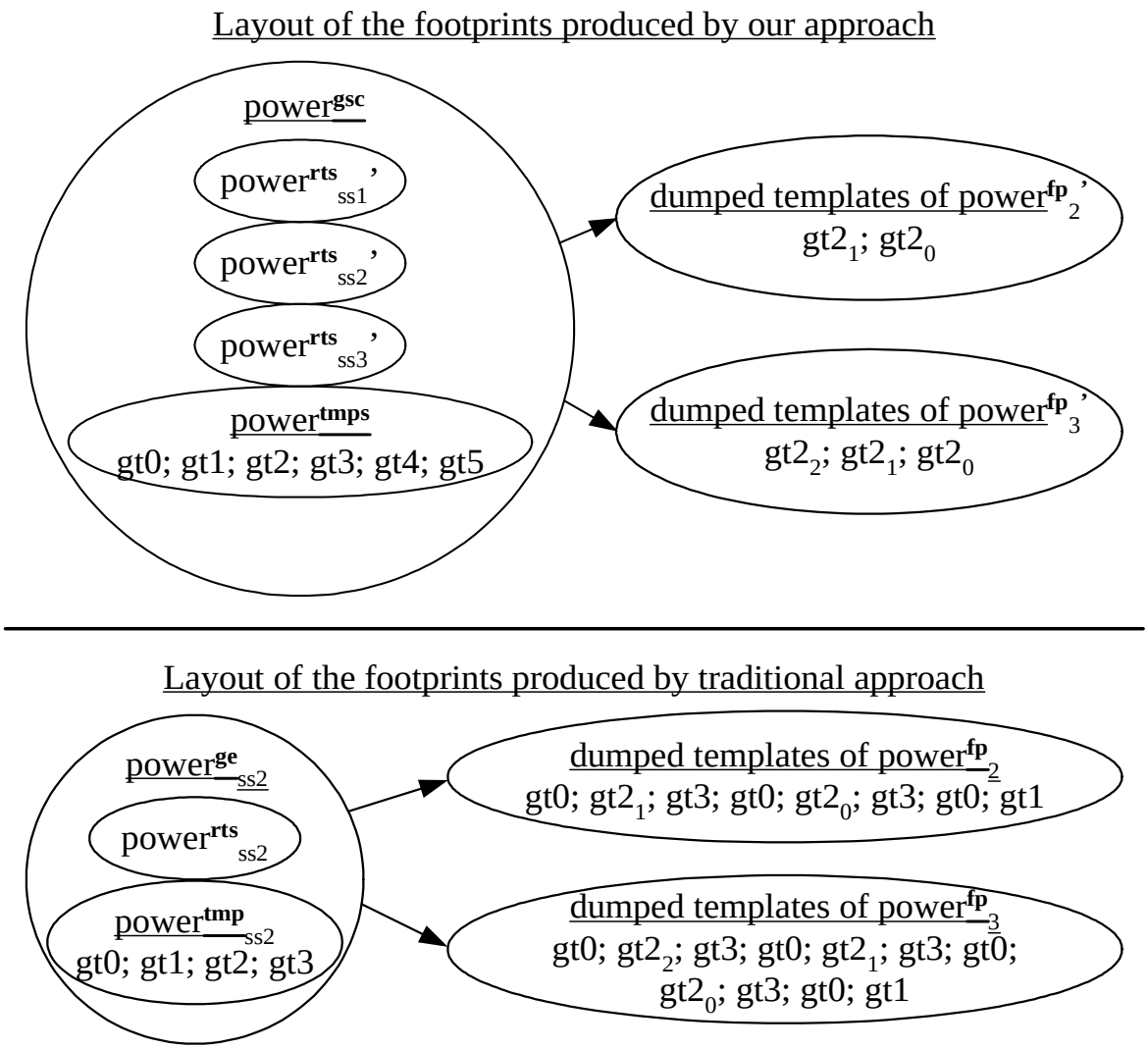

Figure 5.5: Layouts of the footprints of the library `power` with respect to the concrete value 2 produced by our approach and by a traditional approach

The traditional approach allocates a memory block at run-time to store all the

[2]For reasonable comparison, the template files produced by traditional run-time specialization are represented in terms of the templates produced by our approach

templates needed for the footprint. The templates `gt0`, `gt1` and `gt3` stored in the memory blocks allocated for forming the footprints $\texttt{power}_2^{\mathbf{fp}}$ and $\texttt{power}_3^{\mathbf{fp}}$ are identical to the original templates stored in the memory block allocated for the template file of $\texttt{power}_{\texttt{pss2}}^{\mathbf{tmp}}$. These three templates also appear multiple times in $\texttt{power}_2^{\mathbf{fp}}$ and $\texttt{power}_3^{\mathbf{fp}}$.

On the other hand, in our approach, for each footprint the templates are split and kept in two separate memory blocks: (1) A dynamically allocated memory block keeps the instantiated hybrid templates which are dumped by execution of local runtime specializers and instantiated from global template repository $\texttt{power}^{\mathbf{tmps}}$; (2) A memory block keeps the global template repository $\texttt{power}^{\mathbf{tmps}}$. Under this approach, the footprint is produced by *linking templates from the two separate memory blocks*. Referring to Figure 5.5 again, a footprint is constructed by linking the dumped hybrid templates in the dynamically allocated memory block to the totally dynamic templates found in the template repository.

By adopting this template dumping strategy, multiple occurrences of identical templates in the dynamically allocated memory block, whether they originate from a single file or from multiple template files, can be substituted by references to the corresponding single copy of the templates residing in the global template repository of a GSC.

5.2.2 Approach to Connecting Templates

As templates forming a footprint are not laid out in consecutive memory space, we need to connect them together so that execution of the footprint can proceed properly. A naive approach to connecting templates together would be to add a goto instruction at the end of each template jumping to the subsequent template. However we notice that when executing the footprint, even though instantiated hybrid templates residing in the dynamically allocated memory block only need to be executed once, totally dynamic templates residing in the global template repository may need to be executed

multiple times. (Examples are, if a totally dynamic template is nested within a `while` statement and the subsequent template is hybrid, or the original library is a recursive library.) It is possible for a template to be connected logically to many different subsequent templates. Thus it is undesirable to associate with each template a goto instruction to connect them all together.

We tackle this problem as follows:

1. During the first phase at run-time when executing a local runtime specializer, besides creating the footprint (i.e., dumping and instantiating the hybrid templates) we also build an **address table**. This address table records a sequence of addresses of the object templates, depicting the program execution control flow among these templates during the execution of the footprint.

   The address table is constructed by the local run-time specializers, which are aware of the size and location of each object template based on information collected by the template compiler.

2. We add two types of operations for the purpose of passing program execution control among templates. These two operations capture the interactions between object templates and the address table.

   - The **registration operation** whose macro name is `REGISTER`. It registers the address of an object template in the address table. Registration operations are part of a local run-time specializer and are executed during the first phase to build the address table.

     The address of a template is encoded as *the beginning address of the memory + offset of the template to the beginning address of the memory*. Here, the *memory* refers to either dynamically allocated memory or the shared memory of the GSC.

- The **redirecting operation** whose macro name is `REDIRECT`. It redirects the program execution control to the subsequent template at the end of execution of the current template. The address of the subsequent template is found in the address table.

  Except for the templates derived from an action-annotated return statement, redirecting operations are inserted at the end of all templates, including totally dynamic templates and instantiated hybrid templates, so that after reaching the end of a template, execution will flow to the template pointed to by the current address indexed at the address table. Redirecting operations are executed in the second phase to execute the footprint properly.

Suppose the name of the address table is `addr_list`, the name of the template counter which acts as a "program" counter is `tc`, and the address of an object template is `template_addr`, then the `REGISTER` and `REDIRECT` macros are defined in Figure 5.6.

```
int tc;
void ** addr_list;

#define REGISTER (template_addr)
{
  addr_list [tc] = template_addr;
  tc = tc + 1;
}

#define REDIRECT
{
  tc = tc + 1;
  void *addr = addr_list [tc];
  goto *addr;
}
```

Figure 5.6: Design of registration and redirecting operations

The template counter `tc` is initialized to zero at the beginning of the second

phase at run-time. The address of a footprint is the address of the first template to be executed during the second phase of run-time; whether a hybrid template residing in the dynamically allocated memory block or a totally dynamic template residing in the memory block allocated for the global template repository of the GSC.

```
void *power_rts_ss2 (int e) {
  int local_init_tc = tc;
  REGISTER (&gt0);
  if (e == 0)
    REGISTER (&gt1);
  else {
    gt2' = DUMP_TEMPLATE (gt2);
    REGISTER (&gt2');
    PATCH_CALL_HOLE (gt2', CH2, (void *)power_rts_ss2(e - 1));
    power_rts_ss2 (h0);
    REGISTER (&gt3);
  }
  /* return address of first template been registered in this round
  return addr_list[local_init_tc];
}
```

Figure 5.7: The pseudo-code of a local run-time specializer derived from $\texttt{power}^{\mathbf{aa}}_{\mathbf{p_{ss2}}}$

```
gt0 ::= { int z; REDIRECT;}
gt1 ::= { return 1;}
gt2 ::= { tc = tc + 1; z = b * ((int(*)int))(&CH2)(b); REDIRECT;}
gt3 ::= { return z;}
gt4 ::= { if (e ==0) REDIRECT;
          else{tc = tc + 2; void *addr = addr_list[tc]; goto *addr;}
        }
gt5 ::= { tc = tc + 1; z = b * ((int(*)(int, int)))(&CH3)(b, e-1);
          REDIRECT; }
```

Figure 5.8: All distinct templates derived from three action-annotated versions of library `power` (extended version)

In Figure 5.7 we present pseudo-code of a local run-time specializer instrumented with registration operations. It is derived from the action-annotated code $\texttt{power}^{\mathbf{aa}}_{\mathbf{p_{ss2}}}$

depicted in Figure 5.3. Figure 5.8 lists extended versions of the templates listed in Figure 5.1, including redirecting operations.

DUMP_TEMPLATE and PATCH_CALL_HOLE, which are macros of template dumping and instantiating operations respectively, are defined in Figure 5.9. The design is the same as Tempo's.

```
/*
* code_ptr points to dynamically allocated memory block
* tmp_ptr points to beginning address of a template after dumping
* name points to the beginning address of a template before dumping
* size is the size of a template
*/
#define DUMP_TEMPLATE(code_ptr, tmp_ptr, name, size)
{
memcpy(code_ptr,name,size);
tmp_ptr = code_ptr;
code_ptr = code_ptr + size;
}

/*
* e is an expression whose value is used to instantiate call hole
* ho is the offset of call hole within dumped template
*/
#define PATCH_CALL_HOLE(tmp_ptr, e, ho)
{
unsigned long addr = (unsigned long)e;
*((unsigned long *)(tmp_ptr+ho))=(addr-(unsigned long)(tmp_ptr+ho+4));
}
```

Figure 5.9: Design of template dumping and instantiating operations

5.2.3 Functional Specifications of GSC and Its Footprint

Before detailing the algorithm for constructing and executing a GSC, we formulate the compile-time and run-time properties of a GSC and its footprint as follows.

GSC instantiation: $f^{\mathbf{ge}}_{ss} = [\![f^{\mathbf{gsc}}]\!]\ ss$

Footprint construction: $f^{\mathbf{fp}}_{val_s} = [\![f^{\mathbf{rts}}_{ss}]\!]\ val_s$

An equational definition of $f^{\mathbf{gsc}}$:

$$\begin{aligned} [\![f]\!]\ (val_s, val_d) &= [\![f^{\mathbf{fp}}_{val_s}]\!]\ val_d \\ &= [\![\ [\![f^{\mathbf{ge}}_{ss}]\!]\ val_s\ ]\!]\ val_d \\ &= [\![\ [\![\ [\![f^{\mathbf{gsc}}]\!]\ ss\ ]\!] val_s\ ]\!]\ val_d \end{aligned}$$

A GSC $f^{\mathbf{gsc}}$ can thus be deemed as a program generator generator *cogen*, which has been introduced in Subsection 2.3.2, because $f^{\mathbf{gsc}}(ss)$ returns a generating extension for the binding-time signature ss.

5.3 GSC Construction Algorithm

We have developed a modular static transformation algorithm (defined in Figures 5.10 through 5.15) to create a GSC for a library f.

$$\begin{array}{l} \mathcal{GSC}_{fs}\ \in\ (\mathbf{FDef^{aa}})^n \rightarrow \langle \mathbf{Rep_{temp}}\ ,\ (\mathbf{Stat})^n \rangle \\ \\ \mathcal{GSC}_{fs}\ (f^{\mathbf{aa}}_{s_1}, \ldots, f^{\mathbf{aa}}_{s_n}) ::= \\ \quad let\quad \tau_0 = \emptyset \\ \qquad\quad \{sta_{s_i} = (f^{\mathbf{aa}}_{s_i}).\mathbf{body} \mid 1 \leq i \leq n\} \\ \qquad\quad \{\langle \tau_i\ ,\ f^{\mathbf{rts}}_{s_i} \rangle = \mathcal{GSC}_s\ sta_{s_i}\ \tau_{i-1} \mid 1 \leq i \leq n\} \\ \quad in\quad \langle \tau_n\ ,\ (f^{\mathbf{rts}}_{s_1}, \ldots, f^{\mathbf{rts}}_{s_n}) \rangle \end{array}$$

Figure 5.10: Static transformation over action-annotated codes of a library

The input of the main specification $\mathcal{GSC}_{fs}$ (defined in Figure 5.10) is a set of action-annotated libraries $f^{\mathbf{aa}}_{s_1}, \ldots, f^{\mathbf{aa}}_{s_n}$, which are the result of performing action analysis (whose rules are defined in Figures 2.6, 2.7 and 2.8) over the set of binding-time annotated variants produced by profitability analysis. $\mathcal{GSC}_{fs}$ returns a set of local run-time specializers $f^{\mathbf{rts}}_{s_1}, \ldots, f^{\mathbf{rts}}_{s_n}$ corresponding to the respective action-annotated

codes at the input, and a global template repository $\tau \in \mathbf{Rep_{temp}}$. τ is an action-annotated code index table of the type $\mathbf{Stat^{aa}} \rightarrow \mathbf{Stat}$.

The main specification $\mathcal{GSC}_{fs}$ is defined in terms of two auxiliary specifications $\mathcal{GSC}_s$ and $\mathcal{GSC}_e$.

$$\mathcal{GSC}_s \ \in \ \mathbf{Stat^{aa}} \rightarrow \mathbf{Rep_{temp}} \rightarrow \langle \mathbf{Rep_{temp}}, \mathbf{Stat} \rangle$$

$\mathcal{GSC}_s\ (s_1^{\mathbf{aa}}; s_2^{\mathbf{aa}})\ \tau ::=$ Seq-Rule
$\quad let \quad \langle \tau_1\ ,\ rts_1 \rangle = \mathcal{GSC}_s\ s_1^{\mathbf{aa}}\ \tau$
$\qquad\ \ \langle \tau_2\ ,\ rts_2 \rangle = \mathcal{GSC}_s\ s_2^{\mathbf{aa}}\ \tau_1$
$\quad in \quad \langle \tau_2\ ,\ rts_1; rts_2 \rangle$

$\mathcal{GSC}_s\ (\texttt{int}\ v)^{\texttt{id}}\ \tau ::=$ Decl-Rule
$\quad let \quad s^{\mathbf{aa}} = (\texttt{int}\ v)^{\texttt{id}}$
$\qquad\ \ if\ mem\ (\tau,\ s^{\mathbf{aa}})$
$\qquad\ \ then\ temp = get\ (\tau,\ s^{\mathbf{aa}})$
$\qquad\ \ else\ temp = (\ \texttt{int}\ v;\ \texttt{REDIRECT};\ )$
$\qquad\ \ rts = \texttt{REGISTER}\ (\&temp);$
$\quad in \quad \langle \tau \bowtie \{s^{\mathbf{aa}} \mapsto temp\}\ ,\ rts \rangle$

$\mathcal{GSC}_s\ (v = e^{\texttt{id}})^{\texttt{id}}\ \tau ::=$ Ass-Id-Rule
$\quad let \quad s^{\mathbf{aa}} = (v = e^{\texttt{id}})^{\texttt{id}}$
$\qquad\ \ if\ mem\ (\tau,\ s^{\mathbf{aa}})$
$\qquad\ \ then\ temp = get\ (\tau,\ s^{\mathbf{aa}})$
$\qquad\ \ else\ temp = (v = e;\ \texttt{REDIRECT};\ )$
$\qquad\ \ rts = \texttt{REGISTER}\ (\&temp);$
$\quad in \quad \langle \tau \bowtie \{s^{\mathbf{aa}} \mapsto temp\}\ ,\ rts \rangle$

$\mathcal{GSC}_s\ (v = e^{\texttt{ev}})^{\texttt{ev}}\ \tau ::= \langle \tau\ ,\ v = e; \rangle$ Ass-Ev-Rule

$\mathcal{GSC}_s\ (\texttt{return}\ e^{\texttt{ev}})^{\texttt{ev}}\ \tau ::= \langle \tau\ ,\ \texttt{return}\ e; \rangle$ Ret-Ev-Rule

Figure 5.11: Static transformation over an action-annotated statement : Part 1

Description of $\mathcal{GSC}_s$ (defined in Figures 5.11 through 5.14): $\mathcal{GSC}_s$ takes in an action-annotated statement and a global template repository, and returns a (possibly) updated template repository and code forming the local run-time specializer. The template repository operators *mem*, *get*, $\bowtie$ and $\uplus$ used in $\mathcal{GSC}_s$ are defined as follows:

- *mem* $(\tau, s^{\mathbf{aa}})$ returns *true* if τ has an entry with index $s^{\mathbf{aa}}$, otherwise $false$.

$$
\begin{array}{l}
\mathcal{GSC}_s\ (v = e^{\mathbf{aa}})^{\mathtt{rb}}\ \tau ::= \qquad \text{Ass-Rb-Rule} \\
\quad let\ \ \langle e', rts_e\rangle = \mathcal{GSC}_e\ e^{\mathbf{aa}} \\
\qquad\ s^{\mathbf{aa}} = (v = e^{\mathbf{aa}})^{\mathtt{rb}} \\
\qquad\ if\ mem\ (\tau,\ s^{\mathbf{aa}}) \\
\qquad\ then\ temp = get\ (\tau,\ s^{\mathbf{aa}}) \\
\qquad\ else\ temp = (\ v = e';\ \mathtt{REDIRECT};\ ) \\
\qquad\ rts = \ (\ \ rts_e; \\
\qquad\qquad\qquad temp' = \mathtt{DUMP\_TEMPLATE}\ (temp); \\
\qquad\qquad\qquad \mathtt{REGISTER}\ (\&temp');\ ) \\
\quad in\ \ \langle \tau \bowtie \{s^{\mathbf{aa}} \mapsto temp\}\ ,\ rts\rangle \\
\\
\mathcal{GSC}_s\ (\mathtt{return}\ e^{\mathtt{id}})^{\mathtt{id}}\ \tau ::= \qquad \text{Ret-Id-Rule} \\
\quad let\ \ s^{\mathbf{aa}} = (\mathtt{return}\ e^{\mathtt{id}})^{\mathtt{id}} \\
\qquad\ if\ mem\ (\tau,\ s^{\mathbf{aa}}) \\
\qquad\ then\ temp = get\ (\tau,\ s^{\mathbf{aa}}) \\
\qquad\ else\ temp = (\ \mathtt{return}\ e;\ ) \\
\quad in\ \ \langle \tau \bowtie \{s^{\mathbf{aa}} \mapsto temp\}\ ,\ \mathtt{REGISTER}\ (\&temp);\rangle \\
\\
\mathcal{GSC}_s\ (\mathtt{return}\ e^{\mathbf{aa}})^{\mathtt{rb}}\ \tau ::= \qquad \text{Ret-Rb-Rule} \\
\quad let\ \ \langle e', rts_e\rangle = \mathcal{GSC}_e\ e^{\mathbf{aa}} \\
\qquad\ s^{\mathbf{aa}} = (\mathtt{return}\ e^{\mathbf{aa}})^{\mathtt{rb}} \\
\qquad\ if\ mem\ (\tau,\ s^{\mathbf{aa}}) \\
\qquad\ then\ temp = get\ (\tau,\ s^{\mathbf{aa}}) \\
\qquad\ else\ temp = (\ \mathtt{return}\ e';\ ) \\
\qquad\ rts = \ (\ \ rts_e; \\
\qquad\qquad\qquad temp' = \mathtt{DUMP\_TEMPLATE}\ (temp); \\
\qquad\qquad\qquad \mathtt{REGISTER}\ (\&temp');\ ) \\
\quad in\ \ \langle \tau \bowtie \{s^{\mathbf{aa}} \mapsto temp\}\ ,\ rts\rangle
\end{array}
$$

Figure 5.12: Static transformation over an action-annotated statement: Part 2

- $get\ (\tau, s^{\mathbf{aa}})$ retrieves the template saved as an entry of τ with the index $s^{\mathbf{aa}}$.
- $\tau \bowtie \{s^{\mathbf{aa}} \mapsto s\}$ returns a new table τ' which is extended with a new entry $\{s^{\mathbf{aa}} \mapsto s\}$ if $s^{\mathbf{aa}}$ is distinct from all existing indexes of τ. Otherwise, it returns the original τ.

The Seq-Rule dealing with action-annotated sequential statements enables the transformation to descend recursively to the basic program constructs (i.e. `assignment` statements, `local declaration` statement, `return` statements and conditional tests

$\mathcal{GSC}_s$ (`if` $e^{\mathtt{ev}}$ $s_1^{\mathtt{aa}}$ `else` $s_2^{\mathtt{aa}}$)$^{\mathtt{rd}}$ τ ::= If-Rd-Rule

let $\langle \tau_1, rts_{s_1} \rangle = \mathcal{GSC}_s\ s_1^{\mathtt{aa}}\ \tau$
$\langle \tau_2, rts_{s_2} \rangle = \mathcal{GSC}_s\ s_2^{\mathtt{aa}}\ \tau_1$
$rts = (\ \mathtt{if}\ e\ rts_{s_1}\ \mathtt{else}\ rts_{s_2}\ )$
in $\langle \tau_2\ ,\ rts \rangle$

$\mathcal{GSC}_s$ (`if` $e^{\mathtt{rb}}$ $s_1^{\mathtt{aa}}$ `else` $s_2^{\mathtt{aa}}$)$^{\mathtt{rb}}$ τ ::= If-Rb-Rule

let $\langle e', rts_e \rangle = \mathcal{GSC}_e\ e^{\mathtt{rb}}$
$\langle \tau_1, rts_{s_1} \rangle = \mathcal{GSC}_s\ s_1^{\mathtt{aa}}\ \tau$
$\langle \tau_2, rts_{s_2} \rangle = \mathcal{GSC}_s\ s_2^{\mathtt{aa}}\ \tau_1$
if $mem\ (\tau, e^{\mathtt{rb}})$
then $temp = get\ (\tau, e^{\mathtt{rb}})$
else $temp =$ (`if` e'
`REDIRECT;`
`else {`
`tc` = `tc` + *branch_distance*;
`void` * `addr` = `addr_list`[`tc`];
`goto` * `addr`; })
$rts'_e =$ (rts_e;
$temp' =$ `DUMP_TEMPLATE` ($temp$);
`REGISTER` (&$temp'$);)
$rts = (\ rts'_e;\ rts_{s_1};\ rts_{s_2};\ )$
in $\langle \tau_2 \bowtie \{e^{\mathtt{rb}} \mapsto temp\}\ ,\ rts \rangle$

Figure 5.13: Static transformation over an action-annotated statement: Part 3

in `if` or `while` statements) to build templates and local run-time specializers.

The If-Rb-Rule deals with action-annotated `if` statements with their conditional tests annotated by `id` or `rb`. We first derive templates for the two branches, and then build a template for the action-annotated conditional test. The latter template is an `if` statement: Its true branch directs the program execution control to the subsequent template, which is exactly the first template derived for action-annotated branch $s_1^{\mathtt{aa}}$; its false branch directs the program execution control to the first template derived for action-annotated branch $s_2^{\mathtt{aa}}$, where *branch_distance* is the number of templates registered by the local run-time specializer. The value of *branch_distance* is known at compile-time. In this way the template derived for an action-annotated conditional test is properly connected with the templates derived for its action-annotated

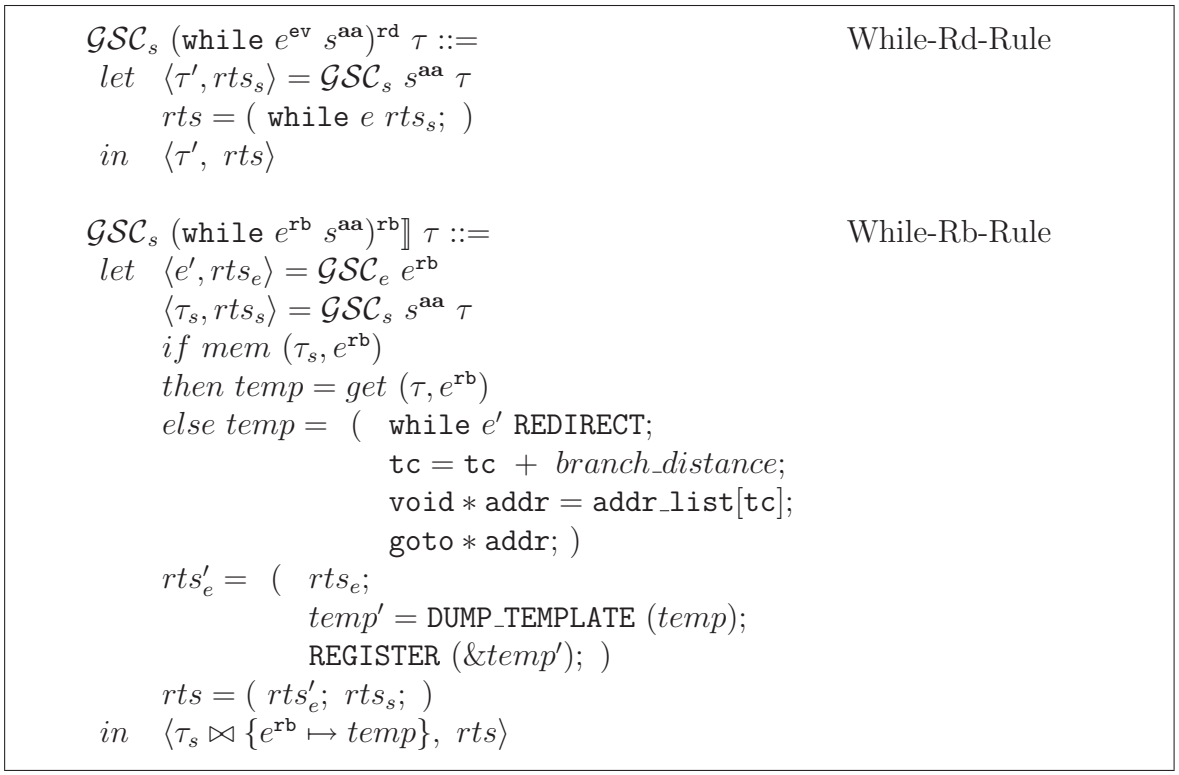

Figure 5.14: Static transformation over an action-annotated statement: Part 4

branches. The If-Rb-Rule enables sharing of templates derived from identical action-annotated statements found in different action-annotated libraries.

The While-Rb-Rule deals with action-annotated `while` statements where the conditional tests are annotated by `id` or `rb`. We first derive templates for the `while` body, and then build a template for the action-annotated conditional test. The latter template is a `while` statement: When the conditional test e' evaluates to true, execution is directed to the subsequent template which is the first template derived for an action-annotated `while` body $s^{\mathtt{aa}}$; when the conditional test e' is evaluated to false, program execution control is directed to the first template after `while` body, where *branch_distance* is the number of templates registered by the local run-time specializer. The value of *branch_distance* is known at compile-time. In this way the template derived for the action-annotated conditional test is properly connected with the templates derived for its action-annotated `while` body. The While-Rb-Rule also

enables program execution to return correctly to the conditional test after executing the final statement of the `while` body.

$$\mathcal{GSC}_e \ \in \ \mathbf{Exp^{aa}} \rightarrow \langle \mathbf{Exp}, \mathbf{Stat} \rangle$$

$\mathcal{GSC}_e\ e^{\mathtt{ev}} ::=$ — Exp-Ev-Rule

$$\begin{array}{ll} \mathit{let} & h \text{ be a fresh hole variable} \\ \mathit{in} & \langle h \ , \ \mathtt{PATCH\_CALL\_HOLE}(h, e) \rangle \end{array}$$

$\mathcal{GSC}_e\ e^{\mathtt{id}} ::= \langle e \ , \ ; \rangle$ — Exp-Id-Rule

$\mathcal{GSC}_e\ (e_1^{\mathbf{aa}} \ \ b_{op} \ \ e_2^{\mathbf{aa}})^{\mathtt{rb}} ::=$ — BinaryExp-Rb-Rule

$$\begin{array}{ll} \mathit{let} & \langle e_1', rts_1 \rangle = \mathcal{GSC}_e\ e_1^{\mathbf{aa}} \\ & \langle e_2', rts_2 \rangle = \mathcal{GSC}_e\ e_2^{\mathbf{aa}} \\ \mathit{in} & \langle (e_1'\ b_{op}\ e_2') \ , \ (rts_1 \ ; \ rts_2) \rangle \end{array}$$

$\mathcal{GSC}_e\ f\ (e_1^{\mathbf{aa}}, \ldots, e_n^{\mathbf{aa}}) ::=$ — FCallExp-Rb-Rule

$$\begin{array}{ll} \mathit{let} & \{\langle e_i', rts_i \rangle = \mathcal{GSC}_e\ e_i^{\mathbf{aa}} \mid 1 \leq i \leq n\} \\ & h \text{ be a fresh call hole variable} \\ & rts_e = \mathtt{PATCH\_CALL\_HOLE}(h, rts\_f(\ldots)) \\ & rts = (rts_1 \ ; \ \ldots \ ; \ rts_n \ ; \ rts_e) \\ \mathit{in} & \langle (\mathtt{tc} = \mathtt{tc} + \mathtt{1}, \ fp) \ , \ rts \rangle \end{array}$$

Figure 5.15: Static transformation over an action-annotated expression

Description of $\mathcal{GSC}_e$ (defined in Figure 5.15): $\mathcal{GSC}_e$ takes in an action-annotated expression, and returns a pair containing code that should appear in the run-time specializer and the template file, respectively. We will explain in particular the rule dealing with a library call in following subsection.

Highlights of our static transformation algorithm: The functionality of our static transformation algorithm is similar to Tempo's algorithm of abstractly interpreting action annotations [25] in that both aim to identify the code that should appear in the run-time specializer and in the template file. The differences between these two algorithms are:

- Our static transformation derives a template at each basic language construct whereas the templates identified by Tempo may include the code derived for a sequence of statements, as shown in Figure 5.2.

- There are two more categories of operations used in our static transformation to manipulate templates, namely registration and redirection, both for the purpose of directing program execution among templates during execution through the help of the address table.

- Our dumping operations are only used in dispatching hybrid templates rather than all templates. The category of a template is clearly indicated by the action annotations of the basic program construct from which the template is derived: a totally dynamic template is derived from a program construct that is annotated as `id`, while a hybrid template is derived from a program construct that is annotated as `rb`.

5.3.1 GSC Construction for Inter-related Libraries

For inter-related libraries, the redirecting operation is not inserted in the templates derived from action-annotated `return` statements, as illustrated by rules Ret-Id-Rule and Ret-Rb-Rule of $\mathcal{GSC}_s$. This is because the program execution control flow for a return statement conforms to the convention of call invocation and return, i.e., the stack discipline.

The FCallExp-Rb-Rule that deals with a library call is defined in Figure 5.15. Here, rts_e is the call (to be made during the first phase when executing local run-time specializers) to the corresponding local run-time specializer rts_f. The rts_f is indexed by the binding-time signature ss which is clearly indicated by action-annotated library call, and is parameterized by static inputs specified in ss. The resulting template contains a comma expression: Before the comma is an expression that increases the template counter `tc` by one; after the comma is a function pointer

fp whose type is the same as rts_f. *fp* points to the address of the corresponding footprint constructed by executing $[\![rts_e]\!]$ (i.e., the call to rts_f). The use of a function pointer is similar to the reference of traditional *specialized function definitions*. The difference is that our technique enables sharing of some common program fragments (i.e., those totally dynamic templates) among the specialized function definitions, while the traditional techniques do not. When such template is executed, the function pointer will ensure the program execution control is transferred to the first template of the pointed footprint, following the convention of handling library call/return.

The rules dealing with action-annotated return statements and library calls ensure that the template counter `tc` obeys the following property throughout run-time execution:

> *Regardless of whichever template is executed,* `tc+1` *always points to the address of the next template to be executed.*

The above property holds regardless of how the next template will be reached. Indeed, a template can be reached by executing either the redirecting operation at the end of the current template, or a function pointer that jumps to the first template of the function body, or a return statement that jumps to the code immediately following the library call. In order to maintain this property, we increase the value of `tc` before executing a function pointer, and refrain from calling redirecting operations during the execution of the return statement.

The values of a call hole within a function pointer template is the address of the first template (either a totally dynamic template or a hybrid template) that is needed to construct the footprint of the corresponding function call. Moreover,

- For a function pointer template translated from an action-annotated library call whose partial arguments are annotated as `rd`, this template must be dumped

into a separate memory region. This is similar to the usual practice of instantiating the function body during call execution.

- For a function pointer template translated from an action-annotated library call whose arguments are all annotated as `id`, this template only needs to be registered by the local runtime specializer. The reason is that at compile-time it is known which template will be chosen at first to construct a footprint. Details can be found by examining the rules If-Rb-Rule and While-Rb-Rule.

  The binding-time signature of an embedded library call is either identical to or different from that of the embedding library definition. We term the former case as *self-recursive specialization*. Depending on whether specialization is self-recursive, the call hole within such function pointer template can be instantiated in following two ways:

  - For a self-recursive specialization, the call hole is instantiated with a call to corresponding local runtime specializer. By doing this, the templates that are needed to construct a footprint for the library call are registered in the address table.
  - Otherwise, the call hole is directly filled with the address of the first template. By doing this, we avoid non-terminating specialization.

    In Figure 5.16 we present the pseudo-code of a local run-time specializers derived from the action-annotated code $\texttt{power}^{\mathbf{aa}}_{\texttt{pss3}}$ depicted in Figure 5.3. `(void *)template_rep_power` is the name of a template repository constructed for the `power` example. It points to the address of the first template.

- The call holes within the two categories of function pointer templates mentioned above are both instantiated using the macro `PATCH_CALL_HOLE` (Figure 5.9).

```
void *power_rts_ss2 (int e) {
  int local_init_tc = tc;

  REGISTER (&gt0);
  REGISTER (&gt4);
  REGISTER (&gt1);
  REGISTER (&gt5);
  PATCH_CALL_HOLE (gt5, CH3, (void *)template_rep_power);
  REGISTER (&gt3);

  /* return address of first template been registered in this round
  return addr_list[local_init_tc];
}
```

Figure 5.16: The pseudo-codes of a local run-time specializer derived from $\texttt{power}^{\mathbf{aa}}_{\texttt{pss3}}$

5.3.2 Footprint Construction for Inter-related Libraries

As demonstrated in $\mathcal{GSC}_s$ and $\mathcal{GSC}_e$, the operations that are responsible for building the address table only exist in the code for the run-time specializers. The code in the templates of a GSC, which is constructed for either an intra-procedural library or an inter-procedural/recursive library, does not include any operations to build the address table.

Specifically, during the process of constructing footprints by executing specialized inter-related (or self-recursive) libraries, only one address table is built throughout the execution of all invoked run-time specializers. Because of the sequential nature of code execution, one template counter is adequate for the role of pointing to the address table and controlling the flow among templates. For example, the address table built after calling the run-time specializer $\texttt{power}^{rts}_{\texttt{pss2}}$ (presented in Figure 5.7) with input 2 during the first phrase at run-time, comprises the following sequence of template addresses:

$$[\&\texttt{gt0};\ \&\texttt{gt2}_1;\ \&\texttt{gt0};\ \&\texttt{gt2}_0;\ \&\texttt{gt0};\ \&\texttt{gt1};\ \&\texttt{gt3};\ \&\texttt{gt3}]$$

where $\texttt{gt2}_1$ and $\texttt{gt2}_0$ are two templates instantiated from their original hybrid template `gt2`, within which the static holes are filled with 1 and 0, respectively.

Interested readers may wish to simulate the sequential execution of the templates listed in Figure 5.7 to verify that the templates are connected properly at run-time with the help of the dynamically-built address table.

5.3.3 Organizing and Compiling Template Repositories

In this subsection, we present in detail how we write valid compilation units for template repositories.

In our approach, each individual template is derived from the basic action-annotated program construct (i.e, `assignment` statements, `local declaration` statements, `return` statements and conditional tests in `if` or `while` statements), *possibly* possessed by different action annotated libraries. The same statement with different action annotations leads to the generation of different templates. The individual templates are assembled to form the body of a template repository as follows (let s be a basic program construct of the original library; $s_{tmp_1}, s_{tmp_2}, \ldots, s_{tmp_n}$ be distinct templates derived from distinct action-annotated statements $s^{\mathbf{aa}}{}_1$, $s^{\mathbf{aa}}{}_2$, ..., $s^{\mathbf{aa}}{}_n$ respectively; `tmp_1_start` / `tmp_1_end`, ..., `tmp_n_start` / `tmp_n_end` be pairs of symbolic labels delimiting each source template):

- if s is an `assignment` statements or `local declaration` statement or conditional tests in `if` or `while` statements), the code looks like:

```
tmp_1_start:
s_tmp1;
tmp_1_end:
  ⋮
tmp_n_start:
s_tmpn;
tmp_n_end:
```

- If s is a `return` statement: since we can not put one return statement after another, as done for the other types of statement shown above, then the code looks like:

```
if N1 {
   tmp_1_start:
   s_tmp1;
   tmp_1_end:
} else if N2 {
   tmp_2_start:
   s_tmp2;
   tmp_2_end:
}
⋮
else {
   tmp_n_start:
   s_tmpn;
   tmp_n_end:
}
```

  where `N1`, `N2` ... are variables of integer type. They are *dummy* variables in the sense that they are used only for the purpose of forming the structure of an `if` statement correctly. Their values are not used at run-time.

To cater for the various specialization scenarios, a template repository is parameterized by *all* parameters of the corresponding original library. On the other hand, as described in Section 5.3.1, the type of the function pointer appearing in a template derived from an action-annotated function call is the same as that of a corresponding local run-time specializer, which is parameterized by static parameters. At the assembly level we know that the first several instructions of the assembly code of a library marshal the parameters (i.e., allocate memory space for the parameters) of that library. These instructions are called the *prologue of a subprogram* in the literature [18]. We resolve the difference in the number of parameters between the local run-time specializer and the template repository as follows:

Let $para_1, \ldots, para_n$ be the parameters of the original library. Then the parameter list of the template repository is written as an array of integers of size n. Suppose the name of the array is $paras$. Then $para_1$ is now referred as $paras$ [0], ..., $para_n$ as $paras$[n-1]. In this way, the parameters of function pointers can be linked correctly with parameters of the template repository.

5.3.4 Wrapped GSC

In order to facilitate the interaction between a GSC and multiple applications, we wrap a GSC with other information, including multiple binding-time signatures and two APIs. These APIs act as interfaces to GSC clients, throughout the entire specialization process from compilation to execution, through reception of binding-time information (from clients) and return of the proper run-time generating extension to be referred to by the specialized application.

Client access to a GSC is mainly guided by submitting a binding-time signature. As a GSC maintains only the run-time specializers associated with the *profitable* binding-time signatures and the *totally dynamic* binding-time signature, client access via binding-time signatures not listed as profitable will have to be converted to profitable ones (or the totally dynamic one) by the wrapped GSC. As mentioned in Chapter 4, selection of the *most appropriate* profitable binding-time signature is guided by the specialization policy, and the selected binding-time signature is called the **minimal profitable context**.

For the convenience of referring to binding-time signatures, binding times `s` and `d` are encoded as the digits 0 and 1 respectively, and a binding-time signature comprising a tuple of binding times is encoded into an integer representing a concatenated string of 0's and 1's. For example, the binding-time signatures `ss1` and `ss2` defined in

Table 4.3 are encoded as 00_2 and 10_2 respectively. We term the value of a binding-time signature as a **binding-time signature value** (BSV for short).

```
interface  f_wgsc {

   private:
       static int      ss_num;
       static int      ss [ss_num];
       static int      temp_rep (...);
       static void     *rts_1 (...);
                    ⋮
       static void     *rts_n (...);          /** n is equal with ss_num

    public:
       int             gsc_bt (int bsv);
       void            *gsc_rtge (int bsv);
}
```

Figure 5.17: Interface of wrapped GSC

The interface of wrapped GSC for a library `f` is defined in Figure 5.17. In the figure,

- `ss_num` is an integer representing the number of binding-time signatures supported by GSC. `ss [ss_num]` is an array of BSVs listing all acceptable (i.e., profitable or totally dynamic) binding-time signatures.
- `static int temp_rep` (...) is the function prototype of the template repository file.
- {`static void` $*$ `rts_i` (...) | `i` $= 1, 2, \ldots,$ `ss_num`}, are function prototypes of the local run-time specializers, each of which is parameterized by the static inputs specified in corresponding binding-time signatures.
- `gsc_bt` takes in a BSV encoding of a binding-time signature, and returns an BSV encoding of a minimal profitable context which will be used in the ensuing

correspondence with this wrapped GSC, in replacement of the input binding-time signature.

- `gsc_rtge` takes in a BSV encoding `bsv` of a binding-time signature that was returned by `gsc_bt` and returns a pointer pointing to the run-time specializer indexed by `bsv`.

  In this way, we allow the interactions between the profitability signature and a GSC to happen before the application link-time, thus making the specialization more effective.

5.4 Experimental Study

We conduct experiments to evaluate the effectiveness of our GSC approach. The benchmark used for experiments is the `power` library. We run the experiments on a 1.99 GHz Intel Core 2 Duo machine with 3-GB main memory. The operating system is Red Hat Linux 3.2.2-5.

Comparison of execution times of a source `power` library and a specialized `power` library:. The specialized `power` library is generated by specializing `power` with respect to a binding-time information "$\texttt{bt}_\texttt{b} == \texttt{d} \wedge \texttt{bt}_\texttt{e} == \texttt{s}$" using our GSC approach. Table 5.2 summarizes the speedups for our GSC approach. Column 2 t_{source} contains execution times of source `power` library. Columns 3 to 6 contain information about our GSC approach that include:

- t_{rts}: It is an execution time of constructing a footprint by executing a run-time specializer at the first phase of run-time.

- t_{spe}: It is an execution time of executing the footprint at the second phase of run-time.

- **Speedup** $= t_{source} \ / \ t_{spe}$

- **Breakeven point**: It is the number of times the footprint must be executed to amortize the cost of generation of the footprint. It is computed as:

$$\textbf{Breakeven point} \;=\; t_{rts} \;/\; (t_{source} \;-\; t_{spe})$$

Parameter values	Source `power`	Specialized `power` generated by our GSC approach			
	t_{source}	t_{rts}	t_{spe}	Speedup	Breakeven point
b = 3; e = 0	3	77	0.03	100	26
b = 3; e = 1	3	31	0.06	50	11
b = 3; e = 2	3	40	0.09	33.33	14
b = 3; e = 3	3	95	0.13	23.08	34
b = 3; e = 4	3	71	0.17	17.65	26
b = 3; e = 5	3	219	0.32	9.38	15
b = 3; e = 6	3	114	0.45	6.67	82
b = 3; e = 7	4	121	0.48	8.33	35
b = 3; e = 8	3	124	0.72	4.17	55
b = 3; e = 9	4	160	1.33	3.01	60
b = 3; e = 10	4	197	1.14	3.51	69
b = 3; e = 11	3	183	1.43	2.10	117
b = 3; e = 12	3	18	1.70	1.76	14
b = 3; e = 13	3	211	2.01	1.49	213
b = 3; e = 14	4	224	2.28	1.75	131
b = 3; e = 15	4	279	2.53	1.58	190

Table 5.2: Comparison of the execution times of unspecialized and specialized `power` generated by our GSC approach (execution times in microseconds)

We find that the specialized `power` library generated by our GSC approach surpasses the source program in terms of execution times, despite the fact that global optimization is disabled when compiling the template repository. Most of the breakeven points that range from 11 to 82 are low.

Comparison of our GSC approach and Tempo: We also compare the execution effectiveness of Tempo and our GSC approach in executing specialized `power` libraries generated by these two approaches, respectively. Table 5.3 summarizes the evaluation results that include:

- s_{fp}: It is the total size of templates dumped by Tempo or GSC approach.
- s_{addr}: It is the size of an address table built by our GSC approach.
- t_{rts}, t_{spe}: Their meanings are the same as those used in Table 5.2.

	Tempo			GSC approach			
Parameter values	s_{fp}	t_{rts}	t_{spe}	s_{fp}	s_{addr}	t_{rts}	t_{spe}
b = 3; e = 0	20	63	0.01	0	12	68	0.03
b = 3; e = 1	62	20	0.03	66	48	30	0.06
b = 3; e = 2	104	21	0.05	198	84	46	0.09
b = 3; e = 3	146	21	0.11	396	120	58	0.13
b = 3; e = 4	188	24	0.11	660	156	111	0.17
b = 3; e = 5	230	22	0.14	990	192	91	0.35
b = 3; e = 6	272	24	0.19	1386	228	110	0.41
b = 3; e = 7	314	24	0.22	1848	264	1576	1.08
b = 3; e = 8	356	25	0.02	2376	300	144	0.77
b = 3; e = 9	398	30	0.62	2970	336	157	1.05

Table 5.3: Comparison of Tempo and our GSC approach (execution times in microseconds, sizes in bytes)

The evaluation result demonstrates that: For this specific benchmark, the execution times and memory usage of our GSC approach are not as good as those of Tempo. There is no chance of a breakeven in terms of memory usage for GSC approach as compared with Tempo. The cause of this situation is that under our current implementation, the size of the codes involving redirecting operations and address tables are around 40 bytes. Thereafter, there is about 40 bytes *additional* of codes for redirecting operations in a hybrid template as compared to the equivalent template in Tempo. For this specific specialized `power` library, the size of a hybrid template, which is involved in constructing the footprint by using our GSC approach, is bigger than the total size of the templates that are involved in constructing the footprint by using Tempo. The savings in space obtained by using our GSC approach is originated from the mechanism of *not dumping the totally dynamic templates*.

The increase of code size plus the generation of address tables by our GSC approach also accounts for the difference of execution times t_{rts} between Tempo and our approach. As for the times of executing footprints generated by Tempo and our GSC approach respectively, the evaluation result shows that there is not much difference between Tempo and our GSC approach in executing templates, which include both hybrid and totally dynamic templates, at the second phase of run-time.

The situation of memory usage and execution times incurred by our GSC approach can be improved in following ways:

- Improve our current implementation of the redirecting operation and address table to reduce the size of codes for redirecting operations.

- Design alternative ways of organizing templates to reduce memory usage at run-time.

  - Our current scheme of organizing and compiling template repository can be improved by grouping adjacent hybrid templates or adjacent totally dynamic templates into a cluster to enable global optimization. In this new schema, there will be only one registration operation and one redirecting operation associated with each cluster.[3] The extra memory usage and run-time overhead caused by executing redirecting operations can thus be more efficiently amortized. Moreover, the new scheme enables some global optimizations to be performed over each cluster of templates to produce optimized object templates, the approach of which is similar to what Tempo and other run-time specialization systems, such as DyC and TickC [66, 65, 4], have done.

[3]The information conveyed by the *web of action annotated codes* [84] regarding the merging of sharable action-annotated code segments can help us to perform the clustering operation.

– The principle of "not dumping total dynamic template" can be compromised by merging some total dynamic templates and their consecutive hybrid templates into one template. Consequently, such merged templates will be treated as hybrid templates and be dumped/instantiated at run-time. This merger trades dumpling of some totally dynamic templates for the reduction in spaces required for additional administrative codes. We would like to investigate a way to decide when total dynamic templates should be merged with the consecutive hybrid templates.

5.5 Summary

In this chapter, we have investigated the techniques used in constructing and executing a generic specialization component (GSC for short) for a library, with the vision of replacing original libraries with GSC that cater for various specialization opportunities, and minimizing the need for repetitive and redundant specialization of libraries at the application level. Specifically, we proposed a static transformation to construct a GSC for a shared library, aiming at eliminating the code duplication occurring at compile-time. Instead of creating separate generating extensions with respect to different binding-time signatures as done by traditional specialization techniques, our GSC is composed of a set of *local* run-time specializers, each of which pertains to a specialization of the library with respect to a specific binding-time signature; and a *global* template repository that is shared by those *local* run-time specializers. Specialization, in addition to analysis, is performed at the library level, decreasing the amount of repeated specialization of the library at the application level. We also proposed a novel run-time specialization approach to minimize the need to dump object templates at run-time to form footprints of GSC and maximize *sharing* by sharing the totally dynamic templates of a GSC among different footprints, at the expense of building an extra address table at run-time.

We have built GSC for several libraries. The source codes of these GSC can be found at the web site

`http://www.comp.nus.edu.sg/~zhuping/prototype/gsc/`

Other researchers have also made efforts in reducing code (and/or computation) duplication of partial evaluation. Related approaches from the literature include:

- Erik Ruf and Daniel Weise [68] explained how to avoid generating redundant specialized code when specializing programs written in a subset of Scheme language. They advocated the computation of a *domain of specialization* of a specialized function definition. A domain of specialization is expressed in terms of the arguments' type information. It is comprised of different sets of argument values for which it is guaranteed to return the same specialized code from the original function definition. This approach captures more opportunities to share the reusable specialized code, compared with a traditional memoization (or cache-based) approach in which the specialized code can be reused only when the values of static parameters match exactly. The idea behind this approach can be exploited in our framework to further reduce code duplication at run-time when creating various instances from hybrid templates.

- Soren Debois [33] devised a compile-time post-processing named *rewinding* to remove duplicated code from a specialized program that is the result of performing loop-invariant code motion and strength reduction over a program written in a flow chart language. The source of code duplication is that specialization is based on a history of execution that is polyvariant. He defined that two basic program blocks are equivalent (and thus redundant) if *"they execute the same assignments, perform the same test, and branch to similarly equivalent blocks"*. Then, code duplication is removed by rewinding the specialized program into a minimal equivalent program.

- Kedar Swadi et al. [75] designed a *monadic combinator* written in multi-stage programming language MetaOCaml. This monad is used to avoid both code duplication and duplication of computation. Our approach is very similar to this approach in the sense that we do not dump all object templates to form the footprint when the values of static parameters are available. Rather, we maintain an *address table* to record the addresses of these templates.

- A *hybrid specialization* approach [8, 34, 47, 48, 49] has been proposed recently. This approach relies on the observation that for some different argument values within some range, the compiled code contains the same instructions and only differ by some constants. Basically, this approach generates generic and highly-optimized templates at compilation-time through exposing some unknown values to the compiler, At run-time specialization is performed for a limited number of instructions in those generic binary templates, i.e. the templates are instantiated with the parameter values. This is a heavyweight approach and involves a complex analysis of the set of specialized code (most probably at the assembly code level) to construct the generic templates.

CHAPTER 6

A FRAMEWORK FOR UNIFYING PROGRAM SLICING AND PARTIAL EVALUATION

The framework for **specialization of applications using shared libraries** elaborated in Chapters 4 and 5 leverages the maturity of existing implementations of specialization techniques, in particular partial evaluation, that have been under development for many years. It would be desirable to enhance various existing specialization techniques, such as partial evaluation and program slicing through cross-fertilization among them. In this chapter, we investigate the relationship between (offline) partial evaluation and (static) program slicing. We build a unified framework that captures the essence of these two specialization techniques and enables a consistent treatment of these two specialization techniques in both forward and backward directions. This unified framework also develops new specialization techniques through cross-fertilization between these two existing specialization techniques.

The rest of this chapter is organized as follows. In Section 6.1 we briefly explain the motivation of this work, introduce the subject language used in this chapter and set the scope of partial evaluation and program slicing to be studied in this work. Section 6.2 presents a theoretical elaboration of the unified framework: We first provide a detailed account of the model used to formulate and compare program slicing and partial evaluation, then we demonstrate how forward program slicing and partial evaluation can be instantiated within the framework, and we next cast backward slicing into the unified framework. In Section 6.3, we discuss the implications deriving from the unified framework and show the benefits of the framework through some

motivating examples.

6.1 Introduction

In an approach similar to design by contract [59], specialization information of a library can be established from the **requirements** and **assertions** established among a library's interfaces interacting with other libraries. A requirement constrains the kind of *input* permissible for invoking a library. On the other hand, an assertion stipulates the kind of *output* behavior acceptable by the calling context.

Current research into library specialization has used two different techniques in exploiting this specialization information: Partial evaluation has been used for library specialization with respect to interface requirements; it propagates input information *forward* to the library's output. Program slicing has been used for library specialization with respect to interface assertions; it performs *backward* specialization which passes information from output back to the library's input. Given the intimate relation between the requirement and the assertion of a library, it is natural to consider the relation between partial evaluation and program slicing, and to explore their potential to enhance existing specialization techniques.

6.1.1 Scope of the Study

As both partial evaluation and slicing are well-developed techniques, they have existed in great variety. In this study we concentrate on comparing the essence of these techniques, and choose a pair of specific variants for comparison to illustrate their commonalities and differences. There are several variants of BTA. In relating partial evaluation and slicing, we consider a variant of BTA called **strong staticness BTA** [30]. **Strong staticness BTA** takes into consideration both the data dependence information and the control dependence information in the program, which is consistent with the idea employed by program slicing. **Strong staticness BTA** will classify a variable as `s` iff it only depends on static variables and constants and it is

not control dependent on any dynamic predicate; otherwise, it is classified as `d`.

We also ignore issues related to specialization termination for the following two reasons: (1) Program slicing does not have a termination problem; and (2) Termination analysis can be considered an add-on to this unified framework.

6.1.2 Subject Language

The subject language used in this chapter is a subset of the language defined in Figure 2.1 excluding function application. Its syntax is defined in Figure 6.1.2.

c	$\in$	**Const**	Numerals or Booleans
v	$\in$	**Var**	Variables
b_{op}	$\in$	**BOp**	Binary operators
	$::=$	$+ \mid - \mid * \mid / \mid == \mid != \mid$ $< \mid > \mid >= \mid <= \mid \&\& \mid \mid\mid$	
e	$\in$	**Exp**	Expressions
	$::=$	$c \mid v \mid e_1\ b_{op}\ e_2$	
s	$\in$	**Stat**	Statements
	$::=$	$s_1; s_2 \mid \texttt{while}\ e\ s \mid v = e \mid \texttt{if}\ e\ s_1\ \texttt{else}\ s_2$	

Figure 6.1: Syntax of the subject language used in Chapter 6

Although function specialization forms a crucial part of partial evaluation, it does not play a central role in the formation of the unified framework. Indeed, we view such function specialization as a refinement of a particular transformation performed by partial evaluation. An imperative language is chosen against other paradigms because of its popularity in the domain of program slicing. Our unified framework can be easily extended to handle other programming paradigms, such as functional programming.

6.2 The Unified Framework

We elect to base our comparison between partial evaluation and program slicing on the characteristics of the *specialized programs* produced by these two techniques. This in turn depends on how decisions are made at each program variable and program point during the respective transformation. We restrict ourselves to relating partial evaluation and *forward* program slicing, as both techniques transform programs forwardly (*ie.*, from program input to output). Even though there is no corresponding backward partial evaluation (except constraint-based ones [36, 44, 50, 53]), we will show that *backward* program slicing can be cast into the same framework as partial evaluation and forward program slicing.

This framework enables us to perceive both program slicing and partial evaluation as three-stage processes, namely: **residual analysis**, **action analysis** and **transformation**.

1. The first stage, **residual analysis**, propagates specialization information throughout the program. It determines the residual information a variable may hold at a program point.

   In off-line partial evaluation, BTA plays such a role; similarly, we define a **forward** (or **backward**) **slicing analysis** for such a role in forward (or backward) static slicing. Specifically, we classify variables at a program point into two categories: **residual** and **transient**. **Residual** variables contribute to the residualization of program points during slicing. The specialization information for forward slicing (i.e., the slicing criterion) defines a set of such **residual** variables at the beginning of the program. On the contrary, the other variables (not included in the slicing criterion) are classified as **transient** at the beginning of the program.

2. The second stage, **action analysis**, uses information derived by residual analysis to determine the action to be taken at each program point. This is similar to the action analysis phase defined in partial evaluators such as Schism [20] and Tempo [22].

   Program slicing treats a statement in two ways: it either removes the statement from the resulting slice or retains the statement. We use the terms **remove** and **retain** respectively to denote these two actions. Partial evaluation transforms program statements in two ways: It either reduces a program construct (an expression or a statement) whose computation is solely based on invariant partial input, but keeps its effect within a partial evaluation environment; or residualizes the program construct whose computation relies on varying input to form the specialized program. We use the terms **reduce** and **residualize** respectively to denote these two actions taken by partial evaluation.

3. The final stage, **transformation**, performs specialization on the program based on the action decisions produced by the action analysis.

6.2.1 Safe Projections

The residual analysis counterparts of partial evaluation and program slicing are the BTA and forward slicing analysis respectively, as introduced above. The relation between BTA and forward slicing analysis can best be described by the notion of **domain projection**, which specifies the capturing of a *certain amount* of information [54].

Definition 6.1 (Launchbury's Domain Projection)**.** *A domain projection* γ *on a domain* **Dom** *is a continuous function* $\gamma : \mathbf{Dom} \rightarrow \mathbf{Dom}$ *such that (i)* $\gamma \sqsubseteq ID$ *and (ii)* $\gamma \circ \gamma = \gamma$ *(idempotent).*

Given a function $f : \mathbf{Dom} \rightarrow \mathbf{Dom}$. Suppose we define a projection γ on **Dom** to obtain a subset of f's results which are of interest. We can then consider the amount

of information on **Dom** that is needed to obtain this γ's result of interests. Let us express this amount using a projection β on **Dom**. Then, the relation between the function f and the projections γ and β possesses the following property, called the **safety condition** [55]:

$$\gamma \circ f = \gamma \circ f \circ \beta$$

A **projection-based analysis** is defined as a program analysis that determines the appropriate projections on program states at each program point, such that these projections and the respective program behaviors, perceived as functions mapping from program states to program states, satisfy the safety condition.

One of our main theses is that: *Both BTA and forward slicing analysis are projection-based analyses on well-classified information. Specifically, BTA is a projection analysis on static information, and forward slicing analysis an equivalent projection-based analysis on transient data.*

6.2.2 Modeling Step-wise Program Behavior

To establish our claim about the relation between partial evaluation and program slicing, we require a model for representing the effect of performing these two specialization techniques on a program. We elect to refine a program model originally proposed by Jones [45], which models a program in terms of its step-wise behavior. This model has been used to define the notion of **congruence**, which has enabled an elegant and intuitive understanding of the correctness of binding-time analysis and partial evaluation.

In Jones' program model, a program p is regarded as a triple of the type $\langle\, \mathcal{P}(\mathbf{PP})$, $\mathcal{P}(\mathbf{Stos})$, $\mathbf{NX}\,\rangle$ where

- **PP** is a set of program points *pp* of integer type. Each program point represents a control point during the computation. A program point is associated with three basic program constructs, i.e., `assignment` statements and conditional

tests of `if` or `while` statements.

- **Stos** is a set of program stores. A program store θ is a set of mappings of type $\mathbf{Var} \rightarrow \mathcal{N} \mid \mathcal{B}$. A program store θ maps variables $x_1, \ldots, x_n$ to their current values $v_1, \ldots, v_n$, and is represented as $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$.

  At each program point pp there may be multiple stores, denoted as $\{\theta_{pp,\ i} \mid 1 \leq i \leq m\}$ for some m.

- **NX** is a function space, each element of which is a step function nx of the type $\langle$ **PP, Stos** $\rangle \rightarrow \langle$ **PP, Stos** $\rangle$. From a program point pp and a program store $\theta_{(pp,\ i)}$, the computation by nx leads to a program point pp' with a program store $\theta'_{pp',\ j}$. More specifically,

  - When an `assignment` statement is executed, the store is updated and program point pp is reset to the immediately successive program point;
  - When a conditional test is executed, only the target program point is updated: After evaluating the conditional test in an `if` statement, the target program point is set to the program point associated with the first statement in the true branch (if the conditional test is evaluated as `true`) or the false branch (if the conditional test is evaluated as `false`); after evaluating the conditional test in a `while` statement, the target program point is set to the program point associated with the first statement in the body of the `while` statement (if the conditional test is evaluated as `true`) or the immediately successive program point of the `while` statement (if the conditional test is evaluated as `false`).

  The program is understood to have terminated with store θ whenever $nx\ \langle pp, \theta \rangle = \langle pp, \theta \rangle$.

The choice of the targeted program point computed by nx depends, in general,

on both the source program point and the program store. Therefore, at a program point pp, we can partition the program store set $stos \in \mathcal{P}(\mathbf{Stos})$ into several subsets $stos_i$ such that if $\theta \in stos_i$, then the targeted program point is pp_i. Furthermore, the new program store at pp_i can be computed by a **control transfer function** ctf_i of type $\mathbf{Stos} \rightarrow \mathbf{Stos}$ on program stores in $stos_i$. That is, $\forall \theta \in stos_i$, $nx\ \langle pp,\ \theta \rangle = \langle pp_i,\ ctf_i\ \theta \rangle$.

A **control transfer** consists of program points pp and pp_i, and its associated function ctf_i, denoted as $\langle pp,\ pp_i,\ ctf_i \rangle$. A **control structure** on a program associates with each program point pp a finite set of control transfers, denoted as $\{\langle pp, pp_i, ctf_i \rangle \mid 1 \leq i \leq m\}$ for some m such that each $stos_i$, which is the domain of ctf_i, is a partition of the complete set of program stores $stos$ at the program point pp into disjoint subsets (i.e., $stos = stos_1 \cup \ldots \cup stos_m$) and for $\theta \in stos_i, nx\ \langle pp,\ \theta \rangle = \langle pp_i,\ ctf_i\ \theta \rangle$.

At this juncture, we introduce two refinements to the above model so that it can capture the essence of specialization modularly.

The first refinement is the relaxation of control transfer. Launchbury pointed out [54] that by defining functions ctf_i above on a particular sub-domain $stos_i$, the control transfer becomes *value dependent*, a condition that is too restrictive for most partial evaluators. He suggests relaxing the domain of ctf_i to $stos$. Consequently, the control transfer accepts value *independent* functions.

The second refinement is the capturing of control dependency in the model. As pointed out by Das [30], the effect of partial evaluation or program slicing does not only depend on static/transient data dependency, but also on dynamic/residual control dependency. Suppose we have the following contrived `if` statement,

During partial evaluation, if the test `(x < 1)` is dynamic, statements at both program points `p3` and `p4` will be residualized. This decision is based on the binding time of the conditional test which is control-related information, not on the value of

Program Points	Program Texts
p1 :	`if (v>1) {`
p2 :	`  if (x<1)`
p3 :	`    v = v+1;`
	`  else`
p4 :	`    v = v-1;`
	`}`

v. The effect of program slicing is similar.

In order to capture such control dependency, we augment a program store to include control-flow information. We define a **control tag** as a boolean value that get its value from the value of a conditional test of either an `if` or a `while` statement. Under this refinement, a refined program store $\theta \in \mathbf{RefStos}$ is represented as a record of the type $\langle \mathbf{vals} : \mathbf{Var} \to \mathcal{N} \mid \mathcal{B},\ \mathbf{ctrs} : \mathbf{CtrStack} \rangle$ where the field **vals** records the program store as usual, and the field **ctrs** records *a stack* of booleans representing nested control tags.

In the above example, a possible program store upon entering the program point `p4` is $\langle \{\mathtt{v} \mapsto 2, \mathtt{x} \mapsto 2\},\ [\mathtt{false}, \mathtt{true}] \rangle$; and upon entering `p3`, it can be $\langle \{\mathtt{v} \mapsto 2, \mathtt{x} \mapsto 0\},\ [\mathtt{true}, \mathtt{true}] \rangle$. In the remainder of this chapter, when we mention program store, we will be referring this refined version.

To precisely reflect the status of nested control tags, we add three extra basic program constructs into the domain **ProgCons**, which otherwise includes `assignment` statements or conditional tests of `if`/`while` statements:

- `exitIf` represents the exit of an `if` statement. Its execution will set the target program point to the program point immediately succeeding the `if` statement.
- `exitWhile` represents the exit of a `while` statement. Its execution will set the target program point to the program point immediately succeeding the `while` statement.

- `endWhilebody` represents the end of the body of a `while` statement. Its execution will set the target program point back to the conditional test of the `while` statement.

Figure 6.2 shows our control transfer function ctf for the subject language defined in Figure 6.1.2.

$$
\begin{array}{lll}
ctf \ \in \ \mathbf{ProgCons} \rightarrow \mathbf{RefStos} \rightarrow \mathbf{RefStos} & & \\
ctf\ (v = e)\ \theta & ::= & let\ r = \mathcal{E}\ e\ \theta \\
 & & in\ \ \theta.\mathbf{vals}[v \leftarrow r] \\
\left.\begin{array}{l} ctf\ (\texttt{if}\ e)\ \theta \\ ctf\ (\texttt{while}\ e)\ \theta \end{array}\right\} & ::= & let\ \ r = \mathcal{E}\ e\ \theta \\
 & & in\ \ \ push\ \ (\theta.\mathbf{ctrs},\ r) \\
\left.\begin{array}{l} ctf\ \texttt{exitIf}\ \theta \\ ctf\ \texttt{exitWhile}\ \theta \\ ctf\ \texttt{endWhilebody}\ \theta \end{array}\right\} & ::= & pop\ \ \theta.\mathbf{ctrs} \\
 & & \\
\mathcal{E} \ \in \ \mathbf{Exp} \rightarrow \mathbf{RefStos} \rightarrow \mathbf{Const} & & \\
\mathcal{E}\ c\ \theta & ::= & c \\
\mathcal{E}\ v\ \theta & ::= & \theta.\mathbf{vals}\ (v) \\
\mathcal{E}\ (e_1\ b_{op}\ e_2)\ \theta & ::= & (\mathcal{E}\ e_1\ \theta)\ b_{op}\ (\mathcal{E}\ e_2\ \theta)
\end{array}
$$

Figure 6.2: Control transfer function ctf over semantic domain

The control transfer function ctf takes in a basic program construct, a program store at the entry of this program construct, and returns a updated program store at the exit of the current program construct. The evaluation function $\mathcal{E}$ evaluates the value of an expression e with respect to its current program store θ. The function $push$ pushes a new control tag r evaluated from the conditional test e into the field **ctrs** of the program store θ; function pop pops the top control tag from the field **ctrs** of the program store θ.

The control transfer functions defined at program constructs `exitWhile` and `endWhilebody` guarantee the finiteness of the stack of control tags even in the presence of infinite loops. In this way we accurately record the change of control tags over every possible execution path.

While the control transfer function needs to update the field of **ctrs** of the program store, it does not rely on the control tag values saved in θ.**ctrs** to compute the values of the program variables at a targeted program point. However, this explicit inclusion of control tags enables us to capture both control dependency and data dependency in the modelling of specialization. This is described in Section 6.2.4.

6.2.3 Congruent Divisions

Jones [12] defines congruence in terms of his definition of control structure and a **program division**. A program division consists of three functions indexed by program points: static (σ), dynamic (δ), and pairing function (π). These three functions must satisfy the following properties: For any $\theta \in$ **Stos**:

$$\pi(\sigma\ \theta,\ \delta\ \theta) = \theta \qquad \sigma(\pi(\theta_s,\ \theta_d)) = \theta_s \qquad \delta(\pi(\theta_s,\ \theta_d)) = \theta_d$$

Intuitively, from the partial evaluation perspective, θ_s ranges over static values, θ_d over dynamic values, and θ over the entire program store. The first condition requires that dividing a program store using static and dynamic functions does not lose any information – the divided information can be reconstructed using the pairing function. The last two conditions require that the static data (constructed by the static function) remains static, and the dynamic date remains dynamic.

The following definition of congruent division is given by Launchbury [54].

Definition 6.2. *A division* (σ, δ, π) *is* congruent at a program point *pp with respect to a control structure* $\{(pp, pp_i, ctf_i : \mathbf{Stos} \to \mathbf{Stos}\}$ *if for each i,*

$$\forall \theta_1, \theta_2 \in \mathbf{Stos} : \sigma_p\ \theta_1 = \sigma_p\ w \quad \Longrightarrow \quad \sigma_{p_i}(ctf_i\ \theta_1) = \sigma_{p_i}(ctf_i\ \theta_2)$$

The implication above is that any two program stores with equal static parts are mapped to new stores whose static parts are also equal. Consequently, the static part of the new stores depends solely on the static parts of the original stores. Thus, if a division is congruent, we will be able to, at partial evaluation, perform computation on static data: At the beginning of the program, the static data are computed from the static input; at every program point pp_i, the static data are derived only from the static data at pp following the control function ctf_i.

Viewing static and dynamic functions as projections over program stores, this construction of congruent division leads to the safety condition of the corresponding projection-based analysis [54]:

Theorem 6.1. *Let (σ, δ, π) be a division at a program point pp with respect to a control structure $\{(pp,\ pp_i,\ ctf_i : \mathbf{Stos} \rightarrow \mathbf{Stos})\}$, then (σ, δ, π) is congruent iff for each i,*

$$\sigma_{pi} \circ ctf_i = \sigma_{pi} \circ ctf_i \circ \sigma_p.$$

6.2.4 Residual Analysis

Both BTA and forward slicing analysis support congruent division of programs. A division supported by BTA has both its σ and δ defined by the computation for static and for dynamic information, respectively. A division supported by forward slicing analysis, on the other hand, has its σ and δ defined by the computation of transient and residual information, respectively.

Figure 6.3 shows our abstract control transfer function $\widehat{ctf}$ for the subject language over an abstract domain. The abstract values belong to a three-value abstract domain $\widehat{\mathbf{Const}}$ consisting of $\{\bot, \triangle, \nabla\}$ with the order $\bot \sqsubset \triangle \sqsubset \nabla$.[1] The abstract program store $\widehat{\theta} \in \widehat{\mathbf{RefStos}}$ is represented as a record of the type $\langle \mathbf{vals} : \mathbf{Var} \rightarrow \widehat{\mathbf{Const}}, \mathbf{ctrs} : \widehat{\mathbf{CtrStack}} \rangle$ where the field **vals** saves the abstract values of the corresponding

[1]Note that the $\bot$ in abstract domain is needed so that the static and dynamic functions (σ and δ) are both well-defined projections on the refined program stores.

variables, the field **ctrs** stores a *stack* of abstract values representing nested control tags. Both are results of evaluating expressions using the abstract evaluation function $\widehat{\mathcal{E}}$.

$$\widehat{ctf} \in \mathbf{ProgCons} \to \widehat{\mathbf{RefStos}} \to \widehat{\mathbf{RefStos}}$$

$$\widehat{ctf}\ (v = e)\ \widehat{\theta} ::= \mathit{let}\ \ r = \widehat{\mathcal{E}}\ e\ \widehat{\theta}\ \ \mathit{ctg} = \mathit{top}\ \ \widehat{\theta}.\mathbf{ctrs}\ \ \mathit{in}\ \ \widehat{\theta}.\mathbf{vals}[v \leftarrow (r \sqcup \mathit{ctg})]$$

$$\left.\begin{array}{l}\widehat{ctf}\ (\texttt{if}\ e)\ \widehat{\theta}\\ \widehat{ctf}\ (\texttt{while}\ e)\ \widehat{\theta}\end{array}\right\} ::= \mathit{let}\ \ r = \widehat{\mathcal{E}}\ e\ \widehat{\theta}\ \ \mathit{ctg} = \mathit{top}\ \ \widehat{\theta}.\mathbf{ctrs}\ \ \mathit{in}\ \ \mathit{push}\ \ (\widehat{\theta}.\mathbf{ctrs},\ r \sqcup \mathit{ctg})$$

$$\left.\begin{array}{l}\widehat{ctf}\ \texttt{exitIf}\ \widehat{\theta}\\ \widehat{ctf}\ \texttt{exitWhile}\ \widehat{\theta}\\ \widehat{ctf}\ \texttt{endWhilebody}\ \widehat{\theta}\end{array}\right\} ::= \mathit{pop}\ \ \widehat{\theta}.\mathbf{ctrs}$$

$$\widehat{\mathcal{E}} \in \mathbf{Exp} \to \widehat{\mathbf{RefStos}} \to \widehat{\mathbf{Const}}$$

$$\widehat{\mathcal{E}}\ c\ \widehat{\theta} ::= \triangle$$

$$\widehat{\mathcal{E}}\ v\ \widehat{\theta} ::= \widehat{\theta}.\mathbf{vals}\ (v)$$

$$\widehat{\mathcal{E}}\ (e_1\ b_{op}\ e_2)\ \widehat{\theta} ::= (\widehat{\mathcal{E}}\ e_1\ \widehat{\theta}) \sqcup (\widehat{\mathcal{E}}\ e_2\ \widehat{\theta})$$

Figure 6.3: Abstract control transfer functions over abstract domain

The abstract control transfer function $\widehat{ctf}$ takes in a basic program construct, an abstract program store at the entry of this program construct, and returns a updated abstract program store at the exit of the current program construct. The abstract evaluation function $\widehat{\mathcal{E}}$ evaluates the value of an expression e with respect to its current abstract program store $\widehat{\theta}$. Function *top* retrieves the top element from the stack saved in $\widehat{\theta}.\mathbf{ctrs}$.

To obtain an abstract control transfer function computing binding-time information, we simply interpret $\triangle$ as static and ∇ to dynamic. To obtain an abstract control transfer function that computes residual information for forward slicing, we instantiate $\triangle$ to transient and ∇ to residual.

The relation between these two analyses can be formally described as follows:

Theorem 6.2. *Both BTA and forward slicing analysis define projections over the control structure* $\{\langle pp,\ pp_i,\ ctf_i\rangle\}$ *at each program point pp such that:*

$$\sigma_{pi}\ \circ\ ctf_i = \sigma_{pi}\ \circ\ ctf_i\ \circ\ \sigma_p.$$

The theorem states that computation of static/transient values depends solely on the other static/transient values.

We have developed a syntax-directed residual analysis $\mathcal{R}$, whose specifications are defined in Figure 6.4.

The residual analysis function $\mathcal{R}$ takes a source statement $s \in$ **Stat** and an abstract program store $\widehat{\theta} \in$ **RefStos**, and returns an residual-information annotated statement denoted by $s^{\widehat{\theta}} \in$ **ResidualStat**. The description of the specification of the residual analysis is as follows:

1. Rule for an `assignment` statement: According to the definition of the abstract evaluation function $\widehat{\mathcal{E}}$ given in Figure 6.3, if the abstract value of the RHS expression is ∇, the abstract value of some variables in this RHS expression must be ∇, i.e. these variables are classified as residual. Thus according to the definition of forward slicing analysis or BTA, the LHS variable that is affected by the *use* of the variables appearing in the RHS expression, is also classified as residual.

2. Rule for `if` and `while` statements: According to the rule dealing with conditional tests of `if` or `while` statements in the abstract control function (defined

$$\mathcal{R} \in \mathbf{Stat} \to \widehat{\mathbf{RefStos}} \to \mathbf{ResidualStat}$$

$$\begin{array}{lcl}
\mathcal{R}\ (v = e)\ \widehat{\theta} & ::= & \mathit{let}\ \ \widehat{\theta}_1 = \widehat{ctf}\ (v = e)\ \ \widehat{\theta} \\
& & \mathit{in}\ \ (v = e)^{\widehat{\theta}_1} \\
\\
\mathcal{R}\ (\texttt{if}\ e\ s_1\ \texttt{else}\ s_2)\ \widehat{\theta} & ::= & \mathit{let}\ \ \widehat{\theta}_1 = \widehat{ctf}\ (\texttt{if}\ e)\ \ \widehat{\theta} \\
& & \quad\ T_1 = \mathcal{R}\ s_i\ \widehat{\theta}_1 \\
& & \quad\ T_2 = \mathcal{R}\ s_i\ \widehat{\theta}_1 \\
& & \mathit{in}\ \ (\texttt{if}\ e^{\theta}\ T_1\ \texttt{else}\ T_2) \\
\\
\mathcal{R}\ (\texttt{while}\ e\ s)\ \widehat{\theta} & ::= & \mathit{let}\ \ \ \widehat{\theta}_1 = \sqcup_{n>0}\ fix\ f\ ^n(\bot)\widehat{\theta} \\
& & \quad\ \ \widehat{\theta'} = \widehat{ctf}\ (\texttt{while}\ e)\ \ \widehat{\theta}_1 \\
& & \quad\ \ T = \mathcal{R}\ s\ \ \widehat{\theta'} \\
& & \mathit{in}\ \ \ (\texttt{while}\ e^{\widehat{\theta}_1}\ T) \\
& & \mathit{where}\ \ fix\ f\ \widehat{\theta}\ = \ \mathit{if}\ \ \ \hat{\mathcal{E}}\ e\ \ \widehat{\theta} = \nabla \\
& & \qquad\qquad\qquad\qquad\ \ \mathit{then}\ \ \widehat{\theta} \\
& & \qquad\qquad\qquad\qquad\ \ \mathit{else}\ \ \ \mathit{let}\ \ T\ =\ \mathcal{R}\ s\ \ \widehat{\theta} \\
& & \qquad\qquad\qquad\qquad\qquad\ \ \mathit{in}\ \ \ f\ (\mathit{getAbsSto}\ T) \\
\\
\mathcal{R}\ (s_1;\ s_2)\ \widehat{\theta} & ::= & \mathit{let}\ \ T_1 = \mathcal{R}\ s_1\ \widehat{\theta} \\
& & \quad\ \widehat{\theta}_1 = \mathit{getAbsSto}\ T_1 \\
& & \quad\ T_2 = \mathcal{R}\ s_2\ \widehat{\theta}_1 \\
& & \mathit{in}\ \ (T_1;T_2)
\end{array}$$

Figure 6.4: Specification of residual analysis $\mathcal{R}$

in Figure 6.3), if the abstract value of the conditional test is ∇, then the abstract value of some variables in this conditional test *or* the enclosing conditional tests must be ∇, i.e. these variables are classified as residual. Since the LHS variables in the two branches of the `if` statement (or in the body of the `while` statement) are (transitively) control dependent on their immediately enclosing conditional test and the nesting conditional tests, the abstract values of those LHS variables should be classified as ∇. The rule dealing with `assignment` statements in the abstract control function (defined in Figure 6.3) guarantees this. If the abstract

value of the conditional test is $\triangle$, we just step into the two branches respectively and update their associated contexts based on current abstract program store. In this way we capture the control dependency between the conditional test and the statements in the two branches of an `if` statement (or in the body of a `while` statement).

The least fix-point of the abstract program store before entering the `while` statement is computed; this is then used to annotate the `while` statement.

3. Rule for sequential statements: This rule shows that in this analysis the specialization information is propagated forwardly, i.e. the state of a statement will affect the states of its successive statements.

$$
\begin{array}{lll}
\multicolumn{3}{l}{getAbsSto \ \in \ \mathbf{ResidualStat} \rightarrow \widehat{\mathbf{RefStos}}} \\
getAbsSto \ \ (v = e)^{\widehat{\theta}} & ::= & \widehat{\theta} \\
getAbsSto \ \ (\texttt{if}\ e^{\widehat{\theta}}\ T_1\ \texttt{else}\ \ T_2) & ::= & let \ \ \widehat{\theta}_1 = getAbsSto \ \ T_1 \\
 & & \quad \widehat{\theta}_2 = getAbsSto \ \ T_2 \\
 & & \quad \langle finalvals, \ \_\rangle = \widehat{\theta}_1 \ \uplus_{\widehat{\theta}} \ \widehat{\theta}_2 \\
 & & \quad finalctrs = \widehat{\theta}.\mathbf{ctrs} \\
 & & in \ \ \langle \mathbf{vals}: \ finalvals, \ \mathbf{ctrs}: \ finalctrs\rangle \\
getAbsSto \ \ (\texttt{while}\ e^{\widehat{\theta}}\ T) & ::= & let \ \ \widehat{\theta}_1 = getAbsSto \ \ T \\
 & & \quad \langle finalvals, \ \_\rangle = \widehat{\theta}_1 \\
 & & \quad finalctrs = \widehat{\theta}.\mathbf{ctrs} \\
 & & in \ \ \langle \mathbf{vals}: \ finalvals, \ \mathbf{ctrs}: \ finalctrs\rangle \\
getAbsSto \ \ (T_1; T_2) & ::= & getAbsSto \ \ T_2
\end{array}
$$

Figure 6.5: Auxiliary function *getAbsSto* used in $\mathcal{R}$

$\mathcal{R}$ depends on an auxiliary function *getAbsSto* (defined in Figure 6.5), which obtains the final abstract program store of a statement. The operator $\uplus_{\widehat{\theta}}$ used in *getAbsSto* is defined as:

- $\widehat{\theta}_1 \ \uplus_{\widehat{\theta}} \ \widehat{\theta}_2 = \langle \ \{v \leftarrow abs_1 \sqcup abs_2 \mid v \mapsto abs_1 \in \widehat{\theta}_1.\mathbf{vals},\ v \mapsto abs_2 \in \widehat{\theta}_2.\mathbf{vals}\},$ $[ctr_i] \mid ctr_i \in \widehat{\theta}_1.\mathbf{ctrs} \cup \widehat{\theta}_2.\mathbf{ctrs} \ \rangle$

The rule dealing with `if` statements in *getAbsSto* says that: The **vals** field of the final abstract program store records the least upper bounds of the abstract values of corresponding variables stored in the final abstract program stores $\widehat{\theta}_1$ and $\widehat{\theta}_2$ of two branches of the `if` statement respectively; the **ctrs** field of the final abstract program store is the same as the **ctrs** field of the abstract program store at the entry of this `if` statement, since the rule dealing with `exitIf` in abstract control function guarantees the abstract values of the conditional tests nested in the two branches, which are pushed in the **ctrs** field of the abstract program store, when dealing with the statements in two branches are *all* popped at the exit of the `if` statement. Similar treatment is defined in the rule dealing with `while` statements in *getAbsSto*

Figure 6.6 demonstrates the residual analysis result on the following contrived code with respect to an abstract program store $\widehat{\theta} = \langle \mathbf{vals} \ : \ \{\mathtt{i} \mapsto \triangle, \mathtt{j} \mapsto \nabla, \mathtt{k} \mapsto \triangle\}, \ \mathbf{ctrs} \ : \ [\,]\rangle$.

Program Text	Residual Analysis Result (Abstract Program States)
`while (i>2) {`	$\langle \mathbf{vals} \ : \ \{\mathtt{i} \mapsto \triangle, \mathtt{j} \mapsto \nabla, \mathtt{k} \mapsto \nabla\}, \ \mathbf{ctrs} \ : \ [\triangle]\rangle$
`  if j > 0`	$\langle \mathbf{vals} \ : \ \{\mathtt{i} \mapsto \triangle, \mathtt{j} \mapsto \nabla, \mathtt{k} \mapsto \nabla\}, \ \mathbf{ctrs} \ : \ [\triangle]\rangle$
`    k=1;`	$\langle \mathbf{vals} \ : \ \{\mathtt{i} \mapsto \triangle, \mathtt{j} \mapsto \nabla, \mathtt{k} \mapsto \nabla\}, \ \mathbf{ctrs} \ : \ [\nabla, \triangle]\rangle$
`  else`	
`    k=2;`	$\langle \mathbf{vals} \ : \ \{\mathtt{i} \mapsto \triangle, \mathtt{j} \mapsto \nabla, \mathtt{k} \mapsto \nabla\}, \ \mathbf{ctrs} \ : \ [\nabla, \triangle]\rangle$
`  i=i-1;`	$\langle \mathbf{vals} \ : \ \{\mathtt{i} \mapsto \triangle, \mathtt{j} \mapsto \nabla, \mathtt{k} \mapsto \nabla\}, \ \mathbf{ctrs} \ : \ [\triangle]\rangle$
`  j=j-1;`	$\langle \mathbf{vals} \ : \ \{\mathtt{i} \mapsto \triangle, \mathtt{j} \mapsto \nabla, \mathtt{k} \mapsto \nabla\}, \ \mathbf{ctrs} \ : \ [\triangle]\rangle$
`}`	

Figure 6.6: An example residual analysis result

6.2.5 Action Analysis and Transformation

We now look at the decisions used by each technique to determine the actions needed at each program point. Here, we associate static variables with transient variables,

and dynamic variables with residual variables. It is then a pleasant surprise to observe that *the decisions for removing/retaining a syntactic construct in program slicing are identical to the decisions for reducing/reconstructing a construct*. That is, both program slicing and partial evaluation have an identical action analysis, modulo the relationship between static/dynamic and transient/residual.

The specification of the action analysis is defined in Figure 6.7.

$$\mathcal{A} \in \mathbf{ResidualStat} \rightarrow \mathbf{AnnStat}$$

$$\begin{array}{lcll}
\mathcal{A}(v = e)^{\widehat{\theta}} & = & if & \widehat{\theta}.\mathbf{vals}(\mathtt{v}) = \nabla \\
 & & then & (\mathtt{v} := e)^{\alpha_1} \\
 & & else & (\mathtt{v} := e)^{\alpha_2} \\
 & & & \\
\mathcal{A}(\mathtt{if}\ e^{\widehat{\theta}}\ T_1\ \mathtt{else}\ T_2) & = & let & U_1 = \mathcal{A}\ T_1 \\
 & & & U_2 = \mathcal{A}\ T_2 \\
 & & in & if \quad \widehat{\mathcal{E}}\ e\ \widehat{\theta} = \nabla \\
 & & & then \quad \mathtt{if}\ e^{\alpha_1}\ \mathtt{then}\ U_1\ \mathtt{else}\ U_2 \\
 & & & else \quad \mathtt{if}\ e^{\alpha_2}\ \mathtt{then}\ U_1\ \mathtt{else}\ U_2 \\
 & & & \\
\mathcal{A}(\mathtt{while}\ e^{\theta}\ T) & = & let & U = \mathcal{A}\ T \\
 & & in & if \quad \widehat{\mathcal{E}}\ e\ \widehat{\theta} = \nabla \\
 & & & then \quad (\mathtt{while}\ e^{\alpha_1}\ U) \\
 & & & else \quad (\mathtt{while}\ e^{\alpha_2}\ U) \\
 & & & \\
\mathcal{A}\ (T_1; T_2) & = & (\mathcal{A}\ T_1; \mathcal{A}\ T_2) &
\end{array}$$

Figure 6.7: Specification for action analysis

The action analysis function $\mathcal{A}$ takes in a residual-analysis information annotated statement $s \in$ **ResidualStat**), and returns a statement $s' \in$ **AnnStat** in which elementary statements are annotated with action-analysis information. The action-analysis information belongs to a set $\mathbf{D_{Act}}$ comprising two meta action variables $\{\alpha_1, \alpha_2\}$. Action α_1 will be instantiated to **retain** in program slicing and **residualize** in partial evaluation. Action α_2 will be instantiated to **remove** in program slicing and **reduce** in partial evaluation.

The result of action analysis, based on the information provided by residual analysis shown in Figure 6.6, is depicted in Figure 6.8.

Program Text	Action Analysis Result
`while (i>2) {`	α_2
`  if j > 0`	α_1
`    k=1;`	α_1
`  else`	
`    k=2;`	α_1
`  i=i-1;`	α_2
`  j=j-1;`	α_1
`}`	

Figure 6.8: An example of action analysis result

The last stage, transformation, produces a specialized program according to the decisions provided by action analysis.

6.2.6 Backward Slicing

While backward slicing has been very popular, there is no corresponding backward technique in typical partial evaluation (except for constraint-based partial evaluation [50]). Nevertheless, we can still cast backward slicing into the unified framework described above.

First and foremost, we observe that forward and backward slicing share *identical* action analysis and the transformation stage. Hence, the only difference lies in their slicing analysis specification. Just like the case of forward slicing, we continue to define those variables declared in the (backward) slicing criterion as **residual variables**, and the other non-declared variables are thus treated as **transient variables**.

Whereas forward slicing ensures that transient values rely solely on other transient values in its computation, backward slicing analysis ensures that *residual values are obtained solely from other residual values*. Indeed, the goal of backward slicing analysis is to deduce the set of variables, at each program point, that must be made

residual in order to support the continual execution leading to the computation of the values of those residual variables at the end of the program. If we define σ and δ as the computation of transient and residual information, respectively, then backward slicing analysis specifies that δ, instead of σ, ensures the congruent division of programs.

Theorem 6.3. *Backward slicing analysis defines a projection over control structure* $\{p - f_i \rightarrow p_i : V \rightarrow V\}$ *at each program point* p *such that:*

$$\delta_{pi} \circ f_i = \delta_{pi} \circ f_i \circ \delta_p.$$

Algorithmically, the analysis for backward slicing analysis will be backward in nature, and thus be distinct from that for forward slicing analysis.

6.3 Benefits of The Framework

In previous sections we have demonstrated that partial evaluation and (forward and backward) program slicing are intimately related, despite the striking semantic differences between the results produced by these techniques. The unified framework theoretically captures the essence of program slicing and partial evaluation. In this section, we show some implications of this unified framework.

6.3.1 Cross-fertilization between Slicing and Partial Evaluation

The value of uniformity between slicing and partial evaluation is not so much that one analysis program may be used for the other, but that the techniques and theories applicable to the one may be used in the other.

Various techniques invented in the past for improving BTA can automatically become candidates for improving forward and backward slicing analysis. These include techniques for bounded static variation and for partially static data. We can obtain a version of backward slicing that handles partially-transient data; this is intimately

related to the backward slicing technique proposed by Reps et al [67]. Vidal et al have demonstrated how to use an existing online partial evaluator to compute (both static and dynamic) slices for logic functional programs [58, 74, 78].

On the other hand, partial evaluation can also be inspired by the ideas in program slicing. For example, the idea of how a constraint is used in backward conditioned slicing [17, 19, 27, 28, 29, 42] (i.e. for each execution path in the program, we associate a corresponding value with respect to the constraint) can be applied in partial evaluation. For another example, the idea of defining (multiple) slicing criteria in the middle of a program and propagating them outwardly leads to the profitability analysis described in Chapter 4, i.e. we can also specify multiple binding-time information in the middle and propagate this information outwardly.

6.3.2 Combining Partial Evaluation and Backward Slicing

We describe here a simple way to combine forward partial evaluation with backward program slicing. That is, we consider specialization of a program with respect to *both* a set of static input variables *and* a backward slicing criterion specified over program outputs.

With two set of specialization contexts to be propagated in opposite directions, we perform two different residual analyses separately to obtain a pair of residual information for each variable: its binding-time value and its residual value. These pairs can be used to drive the action analysis specifically for this specialization. The following table describes their impact:

	static	**dynamic**
transient	**remove**	**remove**
residual	**reduce**	**residualize**

Since a variable with a transient value will not contribute to the construction of

the final specialized program, a statement comprised of only such variables can be safely removed from the specialized program. On the other hand, a variable with static and residual values will be *reduced* by their actual values during specialization, and a variable with dynamic and residual values can be residualized.

Off-line Dynamic Slicing Most existing approaches to dynamic slicing [5, 51, 52] are performed based on user-provided execution history and dynamic dependence graph. Construction of execution history usually requires a high consumption of both space and time, especially for large programs. If we view the input values provided in a dynamic slicing criterion as a form of static information in our framework, the new transformation described above, which combines partial evaluation and backward slicing, provides a fresh perspective on dynamic slicing; we term it **off-line dynamic slicing**.

Off-line dynamic slicing brings the technique of partial evaluation into the realm of program slicing, and replaces the provision of execution history by a partial evaluation process. Being an off-line process, the partial evaluation can be performed very efficiently. For example, given that the dynamic slicing criterion for the following code P is ({i=4; j=1}↓, {k}↑), Figure 6.9 shows the results provided by typical dynamic slicing and offline dynamic slicing.

Original Program	Agrawal's Dynamic Slice	Off-line Dynamic Slice
`while (i>2) {`		
`  if j > 0`		
`    k=1;`		`k=1;`
`  else`		
`    k=2;`	`k=2;`	`k=2;`
`  i=i-1;`		
`  j=j-1;`		
`}`		

Figure 6.9: Example of agrawal's dynamic slice and off-line dynamic slice

In traditional dynamic slicing, the execution history will record the information that the `while` statement is executed twice and the true branch and the false branch of the `if` statement will be chosen at the first and second iterations, respectively. Using a typical dynamic slicing technique (such as that proposed by Agrawal [5]), the false branch will be retained and the true branch removed in the final dynamic slice. On the other hand, using offline dynamic slicing, the `while` statement will be residualized, resulting in two unfoldings of the `while` statement (depicted in Figure 6.9). Finally, through code compression (as typically practised in partial evaluation), only the false branch (i.e. `k= 2`)will be left in the final specialized code.

In this case, the slice thus produced will be identical to that produced by the backward slicing approach proposed by Agrawal [5]. However, in general, the dynamic slice produced by off-line dynamic slicing will be a superset of that produced by Agrawal's approach, since the decision to remove and retain statements is made off-line.

The approach described in this section provides a feasible solution to quasi-static slicing, which was firstly proposed in [77] and aimed to perform slicing in a similar spirit as partial evaluation but remained at the realm of program slicing.

6.4 Summary

The primary role of this chapter is theoretical. We have developed a unified framework to demonstrate that partial evaluation and program slicing can be uniformly defined and compared: We used a refined model, originally proposed by Jones, to represent the small-step behavior of programs; this model enables the co-existence of both static/transient and dynamic/residual data. Based on the model we demonstrated that forward slicing analysis and BTA are both projection-based analysis of the same kind, while the backward slicing analysis is a projection-based analysis over residual data. Interestingly, all three transformations make the same decisions about

transformation actions, modulo the kind of actions chosen.

Based on this unified framework, we demonstrated how partial evaluation and backward slicing can be easily composed to form a new transformation, which mimics the effect of dynamic slicing.

The importance of this result is likely to be application of the uniformity between these two different specialization techniques in work on practical approaches for computing slices or specialized code more efficiently and accurately.

CHAPTER 7

CONCLUSION

In this concluding chapter, we summarize the contribution of this dissertation in Section 7.1 and outline current directions of research in Section 7.2 .

7.1 Summary of the Dissertation

In the last decade, shared libraries are becoming popular commodities for implementing essential services in many systems and application domains. The importance of specialization of application using (shared) libraries has been recognized by the partial evaluation community and substantial progress has been made over the past several years to make partial evaluation feasible in practice. Existing specialization techniques, such as partial evaluation, have been designed for specializing applications using static libraries. When dealing with applications that use shared libraries, the techniques are oblivious to the sharing property of these shared libraries.

In general, specialization of applications using shared libraries can be divided into three sub-problems: (1) independent specialization information generation, which aims to derive specialization information for a library independently, free from the library's deployment contexts, which are usually confined to some specific applications; (2) efficient specialized library construction and execution, the major concern of which is to manage and balance the trade-off between the *multiplicity* of specialized libraries generated with respect to various pieces of specialization information, and the space required for keeping them; and (3) specialization engine enhancement, it is desirable to improve existing specialization techniques through cross-fertilizing different specialization techniques.

This dissertation introduces a comprehensive framework for specialization of applications using shared libraries. The framework consists of three techniques to address the three sub-problems correspondingly.

First, to address the sub-problem of independent specialization information generation, we design a profitability analysis aiming at discovering all meaningful specialization information of a shared library without taking into consideration of its deployment context. Specifically, we advocate the discovery of specialization opportunities by examining the body of the library, and introduce the notion of profitability declaration to capture specialization opportunities independent of how libraries are deployed. This conceptual profitability declaration is translated into a profitability signature which is expressed in the form of the binding-time constraint. A profitability signature stipulates a constraint enforced over library parameters in order to materialize the specialization opportunities within a library.

Second, to address the sub-problem of efficient specialized library construction and execution, we propose a static transformation technique to construct a generic specialization component (GSC for short) for a shared library, aiming at eliminating code duplication occurring at compile-time. Instead of creating separate generating extensions with respect to different binding-time signatures as traditional specialization techniques do, our GSC is composed of a set of local run-time specializers, each of which pertains to a specialization of the library with respect to a specific binding-time signature; and a global template repository that is shared by these local run-time specializers. We also propose a novel run-time specialization approach to minimize the need to dump object templates at run-time and maximize sharing by sharing the totally dynamic templates of a GSC among different footprints, at the expense of building an extra address table at run-time.

Last, to address the third sub-problem of specialization engine enhancement, we develop a unified framework on which partial evaluation and program slicing are

uniformly defined and compared. We use a refined model, originally proposed by Jones, to represent the small-step behavior of a program. This model enables the co-existence of both static/transient and dynamic/residual data. Based on the model, we demonstrate that forward slicing analysis and binding-time analysis are both projection-based analysis of the same kind, while the backward slicing analysis is a projection-based analysis over residual data. Interestingly, all three transformations make the same decisions about transformation actions, modulo the kind of actions chosen. Based on this unified framework, we demonstrate how partial evaluation and backward slicing can be easily composed to form a new transformation, that mimics the effect of dynamic slicing.

Overall, our framework preserves sharing of shared libraries, enables reduction of code duplication during the entire specialization process, and enhances existing specialization techniques through cross-fertilization between program slicing and partial evaluation.

7.2 Research Directions

We have identified the following directions to be pursued in the future.

Specialization of Applications Using Realistic Libraries: In this dissertation we choose a shared library to be a function definition written in a subset of the C language excluding features such as pointers, compound data structures, global variables, etc. We would like to extend the library model to the full C language by including these features since they are common and crucial in the implementation of many system libraries. Correspondingly, the algorithms of our approaches (i.e., profitability analysis, GSC construction and the unified framework for program slicing and partial evaluation, which have been presented in this dissertation) will be refined to cope with those extended features.

Refined specialization techniques: From the perspective of specialization, a typical backward slicing requires minimum information from the specialization information: it simply classifies the variables in a specialization information as either transient or residual. We hope that with more specific specialization information, such as the constancy of some output (transient) variables, a backward specialization will produce a more refined specialized program. Some approaches to backward specialization, such as [67], have exploited static data construction at the output. For general specialization information, we believe that the specialization must be ready to handle *constraints*. There have been multiple works on constraint-based partial evaluation/slicing [17, 19, 27, 28, 29, 36, 42, 44, 50, 53]. In these works, constraints are propagated throughout a program via symbolic predicate transformers to enable aggressive elimination of the branches of `if` statements. This is contrary to the forward specialization which we have described so far, in which the residual information of program variables is represented using two values. It will be interesting to study how to use constraints to enrich and refine the existing three-point domain of the off-line residual analysis.

REFERENCES

[1] Tempo Specializer - User's Manual. Phoenix group. URL: http://phoenix.labri.fr/software/tempo/doc/tempo-doc-user.html.

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

[3] Objective Caml. Phoenix group. URL: http://caml.inria.fr/ocaml/index.en.html.

[4] 'C and tcc. PDOS group, MIT Computer Science and Artificial Intelligence Laboratory. URL: http://pdos.csail.mit.edu/tickc/.

[5] Agrawal, H. and Horgan, J. R., "Dynamic program slicing," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 246–256, 1990.

[6] Andersen, H. M. and Schultz, U. P., "Declarative specialization for object-oriented-program specialization," in *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 27–38, 2004.

[7] Andersen, P. H. and Holst, C. K., "Termination analysis for offline partial evaluation of a higher order functional language," in *Proceedings of International Symposium on Static Analysis*, pp. 67–82, 1996.

[8] Barthou, D., "Contributions to code optimization and high performance library generation," January 2008.

[9] Bhatia, S., *Optimistic Compiler Optimizations for Network Systems.* PhD thesis, University of Bordeaux-I, June, 2006.

[10] Bhatia, S., Consel, C., and Pu, C., "Remote customization of systems code for embedded devices," in *Proceedings of Fourth ACM international conference on Embedded software*, pp. 7–15, 2004.

[11] Binkley, D., Danicic, S., Harman, M., Howroyd, J., and Ouarbya, L., "A formal relationship between program slicing and partial evaluation," *Formal Aspects of Computing*, vol. 18, no. 2, pp. 103–119, 2006.

[12] Bjørner, D., Ershov, A. P., and Jones, N. D., eds., *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop, Gammel Avernaes, Denmark*. October 18-24, 1987.

[13] Bobeff, G. and Noyé, J., "Molding components using program specialization techniques," in *Proceedings of Workshop on Component-Oriented Programming*, July 2003.

[14] Bobeff, G. and Noyé, J., "Component specialization," in *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 39–50, 2004.

[15] Bondorf, A., "Automatic autoprojection of higher order recursive equations," in *Proceedings of European Symposium on Programming*, pp. 70–87, 1990.

[16] Bondorf, A. and Jørgensen, J., "Effiient analyses for realistic off-line partial evaluation: extended version," Tech. Rep. Technical Report Technical Report 93/4, University of Copenhagen, 1993.

[17] Canfora, G., Cimitile, A., and Lucia, A. D., "Conditioned program slicing," in *Information and Software Technology Special Issue on Program Slicing*, vol. 40, pp. 595–607, 1998.

[18] Carter, P. A., *PC Assembly Language*. November 2003. Free ebook.

[19] Chris Fox, Mark Harman, R. M. H. and Danicic, S., "Backward conditioning: A new program specialisation technique and its application to program comprehension," in *Proceedings of Ninth International Workshop on Program Comprehension*, pp. 89–97, 2001.

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

[21] Consel, C. and Danvy, O., "From interpreting to compiling binding times," in *Proceedings of European Symposium on Programming*, pp. 88–105, 1990.

[22] Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S., Volanschi, E.-N., Lawall, J., and Noyé, J., "Tempo: specializing systems applications and beyond," *ACM Computing Survey*, vol. 30, no. 3es, pp. 19–24, 1998.

[23] Consel, C., Hornof, L., Noël, F., Noyé, J., and Volansche, N., "A uniform approach for compile-time and run-time specialization," in *Selected Papers from the Internaltional Seminar on Partial Evaluation*, pp. 54–72, 1996.

[24] Consel, C., Lawall, J., and LeMeur, A.-F., "A tour of tempo: A program specializer for the c language," *Science of Computer Programming*, vol. 52, pp. 341–370, 2004.

[25] Consel, C. and Noël, F., "A general approach for run-time specialization and its application to c," in *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 145–156, 1996.

[26] Cowan, C., Black, A., Krasic, C., Pu, C., Walpole, J., Consel, C., and Volanschi, E.-N., "Specialization classes: an object framework for specialization," in *Proceedings of 5th International Workshop on Object Orientation in Operating Systems*, 1996.

[27] Danicic, S., Daoudi, M., Fox, C., Harman, M., Hierons, R. M., Howroyd, J. R., Ourabya, L., and Ward, M., "Consus: a light-weight program conditioner," *Journal of Systems and Software*, vol. 77, no. 3, pp. 241–262, 2005.

[28] Danicic, S., Fox, C., Harman, M., and Hierons, R. M., "Consit: A conditioned program slicer," in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 216–226, 2000.

[29] Daoudi, M., Ouarbya, L., Howroyd, J., Danicic, S., Harman, M., Fox, C., and Ward, M. P., "Consus: A scalable approach to conditioned slicing," in *Proceedings of 9th Working Conference on Reverse Engineering*, pp. 109–118, 2002.

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

[31] Dean, J., Chambers, C., and Grove, D., "Identifying profitable specialization in object-oriented languages," in *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 85–96, 1994.

[32] Dean, J., Chambers, C., and Grove, D., "Selective specialization for object-oriented languages," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 93–102, 1995.

[33] Debois, S., "Imperative program optimization by partial evaluation," in *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 113–122, 2004.

[34] Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.-T., and Jalby, W., "Exploring application performance: a new tool for a

static/dynamic approach," in *Proceedings of Los Alamos Computer Science Institute Symposium*, October 2005.

[35] Ershov, A. P., "On the essence of compilation," in *Formal Description of Programming Concepts* (Neuhold, E., ed.), pp. 391–420, 1978.

[36] Futamura, Y. and Nogi, K., "Generalized partial computation," in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop, Gammel Avernaes, Denmark* (Bjørner, D., Ershov, A. P., and Jones, N. D., eds.), pp. 131–151, October 18-24, 1987.

[37] Glenstrup, A. J. and Jones, N. D., "Termination analysis and specialization-point insertion in offline partial evaluation," *ACM Transaction on Programming Languages and Systems*, vol. 27, no. 6, pp. 1147–1215, 2005.

[38] Grant, B., Mock, M., Philipose, M., Chambers, C., and Eggers, S. J., "Dyc: an expressive annotation-directed dynamic compiler for c," *Theoretical Computer Science*, vol. 248, no. 1-2, pp. 147–199, 2000.

[39] Grant, B., Philipose, M., Mock, M., Chambers, C., and Eggers, S. J., "An evaluation of staged run-time optimizations in dyc," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 293–304, 1999.

[40] Harman, M., Binkley, D. W., and Danicic, S., "Amorphous program slicing," *Journal of System and Software*, vol. 68, no. 1, pp. 45–64, 2003.

[41] Harman, M. and Danicic, S., "Amorphous program slicing," in *Proceedings of 5th International Workshop on Program Comprehension*, pp. 70–79, 1997.

[42] Harman, M., Hierons, R., Fox, C., Danicic, S., and Howroyd, J., "Pre/post conditioned slicing," in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 138–147, 2001.

[43] Holst, C. K., "Poor man's generalization," August 1998. Working Note, DIKU.

[44] Jin, Y. and Jin, C., "Constraint-based partial evaluation for imperative languages," *Journal of Computer Science and Technology*, vol. 17, no. 1, pp. 64–72, 2002.

[45] Jones, N. D., "Automatic program specialization: A re-examination from basic principles," in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop, Gammel Avernaes, Denmark* (Bjørner, D., Ershov, A. P., and Jones, N. D., eds.), pp. 225–282, October 18-24, 1987.

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

[47] Khan, M. A., Charles, H.-P., and Barthou, D., "An effective automated approach to specialization of code," in *Proceeding of the 20th International Workshop on Languages and Compilers for Parallel Computing*, October 2007.

[48] Khan, M. A., Charles, H.-P., and Barthou, D., "Reducing code size explosion through low-overhead specialization," in *Proceedings of 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*, February 2007.

[49] Khan, M. A., Charles, H.-P., and Barthou, D., "Hybrid specialization: A trade-off between static and dynamic specialization," in *PACT '07: In Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, p. 415, 2007.

[50] Khoo, S.-C. and Shi, K., "Program adaptation via output constraint specialization," *Journal of Higher-Order and Symbolic Computation*, vol. 17 (1-2), pp. 93–128, March - June 2004.

[51] Korel, B. and Laski, J., "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[52] Korel, B. and Laski, J., "Dynamic slicing of computer programs," *Journal of System and Software*, vol. 13, no. 3, pp. 187–195, 1990.

[53] Lafave, L., *A Constraint-based Partial Evaluator for Functional Logic Programs and its Application*. PhD thesis, Department of Computer Science, University of Bristol, 1999.

[54] Launchbury, J., "Strictness and binding-time analyses: two for the price of one," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 80–91, 1991.

[55] Launchbury, J., *Projection Factorisations in Partial Evaluation*. PhD thesis, Department of Computing, University of Glasgow, 1989.

[56] LeMeur, A.-F. and Consel, C., "Generic software component configuration via partial evaluation," in *Proceedings of Product Line Architecture*, August, 2000.

[57] LeMeur, A.-F., Lawall, J., and Consel, C., "Specialization scenarios: A pragmatic approach to declaring program specialization," *Journal of Higher-Order and Symbolic Computation*, vol. 17, no. 1, pp. 47–92, 2004.

[58] Leuschel, M. and Vidal, G., "Forward slicing by conjunctive partial deduction and argument filtering.," in *Proceedings of European Symposium on Programming*, pp. 61–76, 2005.

[59] Meyer, B., "Applying design by contract," *IEEE Computer*, vol. 25, pp. 40–52, October 1992.

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

[61] Mogensen, T. A., "Glossary for partial evaluation and related topics," *Journal of Higher-Order and Symbolic Computation*, vol. 13, no. 4, pp. 355–368, 2000.

[62] Noël, F., *Spécialisation dynamique de code par évaluation partielle*. PhD thesis, Université de Rennes 1, France, Oct. 1996. In French.

[63] Noël, F., Hornof, L., Consel, C., and Lawall, J. L., "Automatic, template-based run-time specialization: Implementation and experimental study," in *In International Conference on Computer Languages*, pp. 132–142, 1996.

[64] Ochoa, C., Silva, J., and Vidal, G., "Program Specialization Based on Dynamic Slicing," in *Proceedings of Workshop on Software Analysis and Development for Pervasive Systems*, pp. 20–31, 2004.

[65] Poletto, M., *Language and compiler support for dynamic code generation*. PhD thesis, MIT, June, 1999.

[66] Poletto, M., Hsieh, W. C., Engler, D. R., and Kaashoek, M. F., "'C and tcc: A language and compiler for dynamic code generation," *ACM Transaction on Programming Languages and Systems*, vol. 21, pp. 324–369, March 1999.

[67] Reps, T. W. and Turnidge, T., "Program specialization via program slicing," in *Selected Papers from the Internaltional Seminar on Partial Evaluation*, pp. 409–429, 1996.

[68] Ruf, E. and Weise, D., "Using types to avoid redundant specialization," in *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 321–333, 1991.

[69] Schultz, U. P., "Black-box program specialization," in *Proceedings of Workshop on Component-Oriented Programming*, 1999.

[70] Schultz, U. P., "Partial evaluation for class-based object-oriented languages," in *Proceedings of the Second Symposium on Programs as Data Objects*, pp. 173–197, 2001.

[71] Schultz, U. P., "Private communication," August 2005.

[72] Schultz, U. P., *Object-Oriented Software Engineering using Partial Evaluation*. PhD thesis, IRISA, University of Rennes I, December 2000.

[73] Schultz, U. P., Lawall, J., and Consel, C., "Automatic program specialization for java," *ACM Transaction on Programming Languages and Systems*, vol. 25, no. 4, pp. 452–499, 2003.

[74] Silva, J. and Vidal, G., "Forward slicing of functional logic programs by partial evaluation," *Theory and Practice of Logic Programming*, vol. 7, no. 1-2, pp. 215–247, 2007.

[75] Swadi, K., Taha, W., Kiselyov, O., and Pasalic, E., "A monadic approach for avoiding code duplication when staging memoized functions," in *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 160–169, 2006.

[76] Tip, F., "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, pp. 121–189, 1995.

[77] Venkatesh, G. A., "The semantic approach to program slicing," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 107–119, 1991.

[78] Vidal, G., "Forward slicing of multi-paradigm declarative programs based on partial evaluation," in *Proceedings of Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR 2002)*, pp. 219–237, 2003.

[79] Volanschi, E.-N., Consel, C., Muller, G., and Cowan, C., "Declarative specialization of object-oriented programs," in *Proceedings of ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pp. 286–300, 1997.

[80] Weiser, M. D., *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, Department of Computer Science and Engineering, University of Michigan, 1979.

[81] Zhu, P. and Khoo, S.-C., "A unified framework for partial evaluation and program slicing," tech. rep., School of Computing, National University of Singapore, 2005.

[82] Zhu, P. and Khoo, S.-C., "Profitability-oriented component specialization," in *Proceedings of Workshop on Component-Oriented Programming*, (Nantes, France), July 2006.

[83] Zhu, P. and Khoo, S.-C., "Request and assert: A pragmatic approach to generating specialization scenarios," tech. rep., School of Computing, National University of Singapore, 2006.

[84] Zhu, P. and Khoo, S.-C., "Towards constructing reusable specialization components," in *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 154–164, 2007.

[85] Zhu, P. and Khoo, S.-C., "Specialization for applications using shared libraries," in *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 159–168, 2008.