THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRB1/05

A Unified Framework for Partial Evaluation and Program Slicing

Ping ZHU and Siau-Cheng KHOO

January 2005

TechnicalReport

Foreword

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

JAFFAR, Joxan
Dean of School

A Unified Framework for Partial Evaluation and Program Slicing

Ping Zhu and Siau-Cheng Khoo

Department of Computer Science
National University of Singapore

January 25, 2005

Abstract

The emphasis on code re-usability in component-based software development has resulted in degradation of system performance. Component specialization techniques have been proposed to overcome this problem. They mainly specialize components wrt contents of component interfaces. This requires specialization to be performed in either a forward or backward direction. The current state of the art applies partial evaluation for forward specialization, but applies backward slicing for backward specialization. In this research, we investigate the relationship between partial evaluation and program slicing in general. We establish a theoretical unified framework in which we can cast both transformations. This framework captures the essence of these two techniques and enables a consistent treatment of program specialization in both forward and backward directions. This framework also make it possible to develop new specialization through cross-fertilization between existing program slicing and partial evaluation techniques.

1 Introduction

The emphasis on the code reusability has raised the genericity in software components, at the expense of run-time efficiency. To address this concern, components are subject to specialization. There are in general two forms of specialization. An *inter*-component specialization attempts to fuse two components into one, thereby eliminating the interface in between. On the other hand, an *intra*-component specialization transforms a component into a more efficient one using information available at the component interface as context for specialization. In this technical report, we focus on intra-component specialization.

Specialization contexts are abstracted from the context in which a component interacts with other components. Specifically, in the "design by contract" approach [32], specialization contexts can be obtained from the *requirements* and *assertions* established among component interfaces. A requirement stipulates the context under which a component will be executed; it constrains the kind of *input* permissible for invoking a component. On the other hand, an assertion stipulates the output behaviour to be expected of the component; it constrains the kind of *output* acceptable by the calling context.

Current research into component specialization has used two different techniques in exploiting these specialization contexts [7, 36]: *Partial evaluation* [24, 25, 22] has been used for component

specialization *wrt* interface requirements; it performs specialization in a *forward* manner that propagates input information towards component output. *Program slicing* [37, 4, 17] has been used for component specialization *wrt* interface assertions; it performs *backward* specialization which passes information from output back to the component's input.

While partial evaluation has been perceived as *the* technique for specialization, the use of program slicing in achieving specialization is intriguing. Given the intimate relation between the requirements and the assertions for a piece of software, it is natural to inquire into the relation between partial evaluation and program slicing, and to explore their potential to improve upon the existing specialization techniques.

In this research, we establish a *unified framework* which captures the essence of both (off-line) partial evaluation and (static) program slicing. To describe the effect of partial evaluation and slicing, we elect to represent program behavior by a small-step model. This model is a refinement of that originally proposed by Jones in [24] for formalizing partial evaluation. We then reformulate typical partial evaluation and forward program slicing using this framework. We demonstrate a close relation between partial evaluation and forward slicing so that these two techniques can be cast in this unified framework. We also demonstrate that backward slicing can be represented similarly using this framework.

This unified framework enables us to assess both specialization techniques in a consistent manner, and to facilitate cross-fertilization between them. Specifically, we demonstrate how a backward slicing and forward partial evaluation can be easily pieced together.

The structure of the technical report is as follows: We discuss related work in Section 2. In Section 3 we describe in relative detail both the existing partial-evaluation and slicing techniques, and introduce the necessary terminology used in this technical report. This is then followed by Section 4 which presents the theoretical elaboration of the unified framework. In Section 5, we show the benefits of this framework through some motivating examples. Before the conclusion, we

explore some future research issues in Section 6.

2 Related Works

Both slicing and partial evaluation are well-developed techniques, and are extensively discussed in the research community. However, there is not much research into integrating these two techniques. The first work, that we know of, relating these two fields is done by Reps and his co-workers, describing a backward program slicing through a projection-based backward analysis [35]. Later, Reps and his student described a partial evaluation using dependence graphs, thus placing both partial evaluation and slicing on the same program representation [14]. However, they do not develop a unified framework to formally investigate the relation between these two techniques, and thus do not produce any new specialization techniques which seamlessly combine the benefits of both partial evaluation and specialization.

There has been a proposal for quasi-static slicing which aims to perform slicing in "a similar spirit as partial evaluation" [38]. However, such a technique remains at the realm of program slicing, and fails to provide a uniform treatment of these two domains of specialization techniques. Moreover, we do not know of any algorithmic description of such a technique.

There is also work on specification-based program slicing (SBS) which aims to refine a program *wrt* pre-postcondition pairs [10, 39]. This approach computes the slice by using predicate transformers to eliminate redundant codes. A code is redundant when it does not affect the constraint, *ie.* its pre-condition and derived post-condition are same. Unfortunately semantics anomaly occurs in SBS, as is depicted in Figure 1.

Here, program P and its slice will produce different values when they are supplied with the same input value. Such a problem occurs frequently when predicate transformers are used to compute imprecise specifications which are then used to identify redundant codes.

```
1. u=x                    1. u=x
2. x=3                    2. x=3
3. w=u                    3. w=u
4. x=w

Program P          SBS of P with slicing criterion
                   ({x>0}, True)
```

Figure 1: Example of semantics anomaly in specification-based slicing

3 Background

In this section, we describe both the existing partial-evaluation and slicing techniques, and introduce the necessary terminology used in this technical report.

The object language used in this technical report is a simple imperative and well-structured language. The abstract syntax of this language is defined in Figure 2.

$$
\begin{array}{rcl}
Pgm & ::= & Stmt^{+} \\
Stmt & ::= & Assign \mid Cond \mid Loop \\
Assign & ::= & Var := Exp \\
Cond & ::= & \texttt{if}\ Exp\ \texttt{then}\ \{Stmt^{+}\}\ \texttt{else}\ \{Stmt^{+}\} \\
Loop & ::= & \texttt{while}\ Exp\ \{Stmt^{+}\} \\
Exp & ::= & Const \mid Var \mid Op\ (Exp, \ldots, Exp) \\
Const & ::= & Num \\
Op & ::= & Aop \mid Bop \mid Rop
\end{array}
$$

Figure 2: The Language Syntax

An imperative language is chosen against other paradigms because of its popularity in the domain of program slicing; the unified framework can be easily extended to handle other programming paradigms, such as the functional paradigm. We exclude function/procedure invocation in the language. Although function/procedure 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 action (called the `residualize` action) performed by partial evaluation.

3.1 Static program slicing

Program slicing is an automatic program extraction technique and it was first proposed by Mark Weiser in his Ph.D thesis[41]. Mark Weiser defined that: A program P' is a slice of program P if P' is a syntactic subset of P and P' is guaranteed to faithfully represent the original program within the domain of specified subset of behavior, which is referred to as a slicing criterion. A slicing criterion specifies a window for observing the behavior of a slice.

Weiser's program slicing has come to be known as *static slicing* , because the slicing criterion contains no information about how the program is to be executed. For static slicing, the slicing criterion is a encoded as a pair $\{i, V\}$ where i is a program point and V is an arbitrary set of variables.

The slicing transformation can treat a statement in two ways: If this statement is included in the final slice, we retain this statement. Otherwise, we remove it from the source program. We use the terms `remove` and `retain` to denote these two actions respectively.

Normally, static slicing can be categorized into forward static slicing [19] and backward static slicing [40]. *Forward static slicing* of a program *wrt* a slicing criterion simply extracts those statements and/or predicates in the program which are affected by the slicing criterion. On the other hand, *backward static slicing* extracts those statements and/or predicates which can have some effect on the slicing criterion.

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

There are three existing basic algorithms to compute a static slice:

1. *Dataflow equations based algorithm*[41]: Slices are computed by iteratively computing consecutive sets of *relevant variables* for each node in the control flow graph (CFG) of the program. This is how Mark Weiser computed slices;

2. *Information flow relations based algorithm*[3]: Each program statement S is associated with three kinds of *Information flow relations*:

   (a) The relation $v\lambda_s e$ can be loosely interpreted as the value of v on entry to S may be used in the evaluation of the expression e in S.

   (b) The relation $e\mu_s v$ signifies that a value of the expression e in S may be used in obtaining the value of the variable v on exit from S.

   (c) The relation $v\rho_s u$ can be loosely interpreted as the value of v on entry to S may be used in obtaining the value of u on exit from S. ρ_s is defined in terms of the previous two relations : $\rho_s = \lambda_s\mu_s \cup \Pi_s$ where $\Pi_s = \{((\mathrm{v}, \mathrm{v}) \in \mathrm{V} \times \mathrm{V}) \mid \mathrm{v} \in \Pi_s\}$ and V is the set of all variables in a program.

   These *Information flow relations* are computed in a syntax-directed, bottom-up manner;

3. *program dependence graph based algorithm*[19]: If a statement is included in the slice, then all its flow-dependence and control-dependence *neighbors* (for *forward slicing*, the *neighbors* refer to the *successors* ; for *backward slicing*, the *neighbors* refer to the *predecessors*) will be included in the slice. This is implemented by forwardly (or backwardly) and transitively traversing over a program representation, called a program dependence graph (PDG)[15, 33].

Forward static slicing has never been used in program specialization, because it *does not* preserve the semantics of the original program. An example of this failure to preserve program semantics is depicted in Figure 3.

```
Program P                  Forward slice of P with slicing
                           criterion (1, {a})

1. x := 1
2. if n>0
3. then
   {
4.   n := n-1
5.   a := a*x               5. a := a*x
6.   while (n>0)
7.   { n := n-1
8.     a := a*x }           8. a := a*x
   }
```

Figure 3: Example of semantics anomaly in forward slicing

Here, a program P is sliced *wrt* the slicing criterion (1, {a}). The resulting slice will not compute the same result as the original program. In fact, the slice may not even be a proper program, as variable x becomes ill-defined.

The reason that the semantics anomaly exists in forward static slicing is that it only captures the computation threads affected by the variables specified in slicing criterion, which are not the same as those computation threads that contribute to the final state of those variables.

On the other hand, the *backward slice* is semantically equivalent to the source program *wrt* the variables specified in the slicing criterion. This semantically preservation is justified in [34].

3.2 Offline partial evaluation

Partial evaluation is a well-developed program specialization technique. Kleene's s-m-n theorem (1952) establishes the mathematical formulation of partial evaluation. N. D. Jones defines partial evaluation as a process which, "when given a program and some of its input data, produces a so-called residual or specialized program. Running the residual program on the remaining input data will yield the same result as running the original program on all of its input data." [25] Compared with the traditional one-stage program execution, partial evaluation is a two-stage program

execution, which can be formulated as follows:

> Given a program P with two inputs in_1 and in_2. If $P_1 = [\![PE]\!][p, in_1]$ then $[\![P]\!][in_1, in_2] = [\![P_1]\!][in_2]$, provided that all the computations involved terminate.

Here, the notion $[\![.]\!]$ signifies the conversion of a program into a process. $[\![PE]\!]$ denotes the partial-evaluation process. The in_1 is available during partial-evaluation time and is termed the *static* data. On the other hand, the in_2, which is absent during partial-evaluation time, is termed the *dynamic* data.

The efficiency of the original program is improved by evaluating the expressions in the program that depend on the fixed input and generating specialized code for those expressions that depend on the run-time input data. It often happens that the time used in two-stage program execution is less than that used in one-stage program execution. For example, compilation plus target run time is often faster than interpretation. At any rate, partial evaluation provides a linear improvement: it does not change the complexity of the analysis [1].

A partial-evaluation process is required to make the following decision on each program statement: Whether the operation applied to the statement should be `reduce` or `residualize`. The `reduce` decision will remove a statement from the residual program, but keeping its effect within a partial-evaluation environment. The `residualize` decision attempts to use the static data, kept in the environment, to reconstruct the statement. Simply put, `reduce` is to simplify an expression, so that it will take less steps to compute in the future, `residualize` is to freeze the expression for future execution.

According to the time at which these decisions are made, we can categorize partial evaluation into online partial evaluation and offline partial evaluation (offline PE). An offline PE will typically involve a preprocessing phase called *binding-time analysis* (BTA). This attempts to determine the binding-time of each variable at each program point. The binding time of a variable asserts that the variable will possess a specific value during partial-evaluation time, if so it is classified as *static*,

otherwise it is classified as *dynamic*. It is computed by propagating the binding-time knowledge of the program's input forwardly.

There are several variants of BTA. In relating partial evaluation and slicing, we consider a variant of BTA called "Strong Staticness BTA" [14], as this analysis is intimately related to the analysis performed by forward static slicing. Strong staticness BTA will classify a variable as *static* if and only if its data only depends on *static* variables and constants and it is not control dependent on any *dynamic* predicate; otherwise, it is classified as *dynamic*. Strong staticness BTA differs from the conventional BTA by taking into consideration the control dependency information in the program, which is consistent with the idea in program slicing.

A statement can be `reduced` only when all variables involved are classified as *static*, otherwise, it should be `residualized`.

4 The Unified Framework

We first 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 the constraint-based ones), in Section 4.4 we will show that *backward* program slicing can be cast into the same framework as the other transformations.

As both partial evaluation and slicing are well-developed techniques, they have existed in great variety. We therefore concentrate on comparing the essence of these techniques, and choose a pair of specific variants for comparison to illustrate their commonalities and differences. 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.

4.1 Three-stage Transformation

We decompose the transformation processes into three stages in a pipeline, namely: *residual analysis*, *action analysis* and *transformation*. This is depicted in Figure 4.

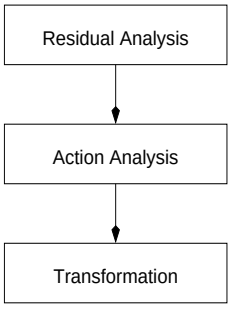

Figure 4: Unified Framework

The residual analysis propagates the specialization context throughout the program. It determines the kind of information a variable may hold at a program point. In off-line partial evaluation, binding-time analysis plays such a role; similarly, we define a (*forward*) *slicing analysis* for such a role in static forward 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 context for a forward slicing (*ie.*, the slicing criterion) defines a set of such residual variables at the beginning of the program. The other variables (not included in the slicing criterion) are classified as transient at the beginning of the program.

Action analysis uses information provided by residual analysis to determine the actions to be taken at each program point. This is similar to the action analysis phase defined in partial evaluators such as Schism [11] and Tempo [12]. It suffices to set the actions taken by partial evaluation at a program point to be either `reduce` or `residualize`; the actions taken by a slicing transformer can either be `remove` or `retain`, as described in Section 3.

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

4.2 Uniformity in Residual Analysis

We claim that both binding-time analysis and forward slicing analyses are projection-based analysis on well-classified information. Specifically, *binding-time analysis is a projection analysis on static information, and forward slicing analysis an equivalent projection-based analysis on transient data*. This is one of the main theses in this technical report.

4.2.1 Safe Projections and Congruent Divisions

The relation between binding-time analysis and forward slicing analysis can best be described by the notion of *domain projection*, which specifies capturing of a "certain amount" of information [31].

Definition 1 *A domain* projection γ *on a domain* D *is a continuous function* $\gamma : D \to D$ *such that (i)* $\gamma \sqsubseteq ID$ *and (ii)* $\gamma \circ \gamma = \gamma$ *(idempotence).*

Given a function $f : D \to D'$. Suppose we define a projection γ on D' to obtain a subset of f's results which are of interest. We can then enquire the amount of information needed at D that is needed to obtain this γ's worth of result. Let us express this amount using a projection β on D. Then, the relation between the function f and the projection γ and β possesses the following property [30], called the *safety* condition:

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

A projection-based analysis is thus 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.

4.2.2 Modelling Step-Wise Program Behavior

To establish the claim about the uniformity between binding-time analysis and forward slicing analysis, we require a model for representing the effect of partial evaluation and program slicing on a program. We elect to refine a program model originally proposed by Jones [24], which models a program in terms of its step-wise behavior. This model has been used to define the notion of *congruence*, which provides an elegant and intuitive understanding of the correctness of binding-time analysis and partial evaluation.

In the Jones' model, a program is regarded as a triple (P, V, nx) where P is a set of program points, V a set of values (states) and nx a step function mapping pairs of (p, v) into pairs of (p', v'). Each (p, v) pair represents a program point in the computation, and the function nx defines both a single computation step and a control-transfer function: From a program point p and program state v, the computation by nx leads to a program point p' with a program state v'. The program is understood to have terminated with value v whenever $nx(p, v) = (p, v)$.

The choice of the targeted program point computed by nx depends, in general, on both the source program point and the program state. Therefore, at a program point p, we can partition the program-state set V into subsets $\{V_i\}$ such that if $v \in V_i$, then the targeted point program is p_i. Furthermore, the new program state at p_i can be computed by a function f_i on program state in V_i. That is, $\forall\ v \in V_i$, $nx(p, v) = (p_i, f_i\ v)$. Such a choice of partition and associated function is called a *control transfer*. Finally, a collection of control transfers associated with a program point is called a *control structure*, denoted by $\{(V_i, p - f_i \rightarrow p_i : V_i \rightarrow V)\}$.

At this juncture, we introduce two refinements to the above model to capture the essence of specialization more accurately.

The first refinement is the relaxation of the definition of control transfer. Launchbury points out, in [31], that by defining functions $\{f_i\}$ above on a particular sub-domain V_i, it only makes sense to draw the values v and w from V_i, and the control transfer becomes *value dependence*,

a condition that is too restrictive for most partial evaluators. He suggests relaxing the domain specification of $\{f_i\}$ so that we have $(V_i, p - f_i \rightarrow p_i : V \rightarrow V)$ for each i. Consequently, the control transfer accepts value *independence* functions.

The second refinement is the capturing of control dependency in the model. As pointed out by Das [14], the effect of partial evaluation and program slicing does not only depends on static/transient data dependency, but also depend on dynamic/residual control dependency. Suppose we have the conditional:

```
        if (v > 1) then
           if (x < 1) then
p1:             v = v + 1
           else
p2:             v = v - 1
```

During partial evaluation, if the test `(x < 1)` is dynamic, statements at both program points `p1` and `p2` will be residualized. This decision is based on the dynamicity of the test, not on value of the `v`. The effect of program slicing is similar.

In order to capture such control dependency, we augment the set of program states to include control-flow information. Specifically, we define a *control state* as a boolean value evaluated from either a conditional test or a while test. A program state θ ($\in$ V) is a tuple consisting of values associated with each of the program variables, and *a stack of booleans* representing nested control states. In the above example, we can represent its program states by the tuple (v, x, cs), where v and x are the values for program variables `v` and `x` respectively, and cs is a stack of control states. A possible program state upon entering program point `p2` is $(2, 2, [\mathit{false}, \mathit{true}])$; and upon entering `p1`, it can be $(2, 0, [\mathit{true}, \mathit{true}])$.

The enrichment of program states can be viewed as an instrumentation of the original program model proposed by Jones. In Figure 5, we show the control transfer function for the language defined in Figure 2. We assume that a program state contains values of variables `x`, `y` and `w`, and

a stack of control states.

$p_1:$	$[\![w = e]\!]$	$f(x, y, w, cs)$
		$= \ let\ r = \mathcal{E}[e][x/\mathtt{x}, y/\mathtt{y}, w/\mathtt{w}]$
		$\quad in\ (x, y, r, cs)$
$p_1:$	$[\![if\ e]\!]$	$f(x, y, w, cs)$
		$= \ let\ r = \mathcal{E}[e][x/\mathtt{x}, y/\mathtt{y}, w/\mathtt{w}]$
		$\quad in\ (x, y, w, r : cs)$
$p_1:$	$[\![endIf]\!]$	$f(x, y, w, c : cs)$
		$= \ (x, y, w, cs)$
$p_1:$	$[\![while\ e]\!]$	$f(x, y, w, cs)$
		$= \ let\ r = \mathcal{E}[e][x/\mathtt{x}, y/\mathtt{y}, w/\mathtt{w}]$
		$\quad in\ (x, y, w, r : cs)$
$p_1:$	$[\![endLbody]\!]$	$f(x, y, w, c : cs)$
		$= \ (x, y, w, cs)$
$p_1:$	$[\![endWhile]\!]$	$f(x, y, w, c : cs)$
		$= \ (x, y, w, cs)$

Figure 5: Control-Transfer Function $p - f \rightarrow p_1$ over semantic domain

In the figure, $\mathcal{E}$ is the semantic function of the language. The constructs $[\![endIf]\!]$, $[\![endLbody]\!]$ and $[\![endWhile]\!]$ represent program points at the end of conditional statement, the end of Loop body (the control will loop back to the `while` test), and the end of Loop statement (the control will go out of `while` statement next) respectively. The control-transfer functions defined at $[\![endLbody]\!]$ and $[\![endWhile]\!]$ guarantee the finiteness of the stack of control state even in the presence of infinite loops. In this way we accurately record the change of control states over every possible execution path.

While the control-transfer function need to be updated to maintain the stack of control states, it does not rely on the stack value to compute the values of the program variables at a targeted program point. However, this explicit inclusion of control states enables us to capture both control dependency and data dependency in the modelling of specialization. This is described in Section 4.2.4.

4.2.3 Congruent Divisions

Jones defines congruence in terms of a control structure and a program *division*. A division consists of three collections of functions indexed by program points. These three collections of functions are: static (σ), dynamic (δ), and pairing (π). Formally,

Definition 2 *A* division *over a program state set V consists of three functions, static (σ), dynamic (δ), and pairing (π), such that for any $v \in V$:*

$$\begin{aligned} \pi(\sigma\ v, \delta\ v) &= v \\ \sigma(\pi(v_s, v_d)) &= v_s \\ \delta(\pi(v_s, v_d)) &= v_d \end{aligned}$$

Intuitively, from the partial evaluation perspective, v_s ranges over static values, v_d over dynamic values, and v over the entire program-state set V. The first condition requires that dividing a program state using static and dynamic functions does not lose any information – the divided information can be resconstructed using the pairing function. The last two conditions require that the static data (constructed by the static function) remains static, and the dynamic remains dynamic.

The following definition of congruence division is given by Launchbury in [31].

Definition 3 *A division (σ, δ, π) is* congruent at a program point p *wrt a control structure $\{V_i, p - f_i \to p_i : V \to V\}$ if for each i,*

$$\begin{aligned} \forall v, w \in V. \quad & \sigma_p\ v = \sigma_p\ w \\ \implies \quad & \sigma_{pi}(f_i\ v) = \sigma_{pi}(f_i\ w) \end{aligned}$$

The implication above states that any two values with equal static parts are mapped to new values whose static parts are also equal. Consequently, the static part of the new values depends solely on the static parts of the original. Thus, if a division is congruent, we will be able, at partial evaluation, to perform computation on the static data: At the beginning of the program, the static

data is computed from the static input; at every program point p_i, the static data will be derived only from the static data at p following the control function $p - f_i \rightarrow p_i$.

Viewing the static and dynamic functions as projections over program states, this construction of congruent division leads to the safe condition of the corresponding projection-based analysis [31]:

Theorem 1 *Let (σ, δ, π) be a division at a program point p wrt a control structure $\{V_i, p - f_i \rightarrow p_i : V \rightarrow V\}$, then (σ, δ, π) is congruent if and only if for each i,*

$$\sigma_{pi} \circ f_i \;=\; \sigma_{pi} \circ f_i \circ \sigma_p.$$

4.2.4 Residual Analysis

The residual analysis component of partial evaluation and slicing are the binding-time analysis and forward slicing analysis respectively. Both analyses support congruent division of programs. A division supported by binding-time analysis 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.

$$
\begin{array}{lll}
p_1: & [\![w = e]\!] & \hat{f}(x, y, w, c : cs) \\
 & & = \; let \quad r = \hat{\mathcal{E}}[e][x/\mathtt{x}, y/\mathtt{y}, w/\mathtt{w}] \\
 & & \quad\; in \quad (x, y, r \sqcup c, c : cs) \\
p_1: & [\![if\ e\ then]\!] & \hat{f}(x, y, w, c : cs) \\
 & & = \; let \quad r = \hat{\mathcal{E}}[e][x/\mathtt{x}, y/\mathtt{y}, w/\mathtt{w}] \\
 & & \quad\; in \quad (x, y, w, r \sqcup c : c : cs) \\
p_1: & [\![endIf]\!] & \hat{f}(x, y, w, c : cs) \;=\; (x, y, w, cs) \\
p_1: & [\![while\ e]\!] & \hat{f}(x, y, w, c : cs) \\
 & & = \; let \quad r = \hat{\mathcal{E}}[e][x/\mathtt{x}, y/\mathtt{y}, w/\mathtt{w}] \\
 & & \quad\; in \quad (x, y, w, r \sqcup c : c : cs) \\
p_1: & [\![endLbody]\!] & \hat{f}(x, y, w, c : cs) \;=\; (x, y, w, cs) \\
p_1: & [\![endWhile]\!] & \hat{f}(x, y, w, c : cs) \;=\; (x, y, w, cs)
\end{array}
$$

Figure 6: Abstract Control-Transfer Functions $p - \hat{f} \rightarrow p_1$ over abstract domain

Figure 6 defines the abstract control-transfer function $\hat{f}$ for the language. It is defined in

$\hat{\mathcal{E}}[const]\theta = \underline{\natural}$
$\hat{\mathcal{E}}[var]\theta = \theta|_{var}$
$\hat{\mathcal{E}}[op\ (e_1, \ldots, e_n)]\theta = \bigsqcup(\hat{\mathcal{E}}[e_i]\theta)\ where\ i = 1, \ldots, n$

Figure 7: Abstract Evaluation Function $\hat{\mathcal{E}}$ over abstract domain

terms of the abstract evaluation function $\hat{\mathcal{E}}$, which is defined in Figure 7. These functions are parameterized by a three-point lifted domain consisting of $\{\bot, \underline{\natural}, \overline{\natural}\}$ such that $\bot < \underline{\natural} < \overline{\natural}$. To obtain an abstract control-transfer function computing binding-time information, we simply instantiate $\underline{\natural}$ to *static* and $\overline{\natural}$ to *dynamic*. To obtain an abstract control-transfer function that computes residual information for forward slicing, we instantiate $\underline{\natural}$ to *transient* and $\overline{\natural}$ to *residual*.[1]

The rule of assignment in Figure 6 demonstrates that stack values have effect on computing the abstract values of the program variables. In this way, the control dependence information is incorporated into the computation of the residual information in binding time analysis and forward slicing analysis.

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

Theorem 2 *Both binding-time analysis and forward slicing analysis define a projection over a control structure* $\{p - f_i \rightarrow p_i : V \rightarrow V\}$ *at each program point* p *such that:*

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

We also develop a syntax-directed specification for residual analysis which is depicted in Figure 8 in Appendix A. It is easy to show that this specification can be constructed from an abstraction of the composition of the abstract control-transfer functions defined in Figure 6. We also present a short description about this specification in Appendix A to show that this specification propagates the data and control dependence information through the program in the same way as how forward slicing analysis and strong staticness BTA do.

[1]Note that the abstract domain is lifted so that the static and dynamic functions (σ and δ) are both projections.

4.3 Action Analysis and Transformation

Here, we associate *static* variables with *transient* variables, and *dynamic* variables with *residual* variables. It is satisfying to observe that *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 identical action analysis, modulo the equivalency between static/dynamic and transient/residual.

The specification for action analysis is depicted in Appendix B Figure 9. Just like the residual analysis, this specification is parameterized by a set of actions, denoted by $\{\alpha_1, \alpha_2\}$. The action α_1 will be instantiated as `retain` in program slicing and `residualize` in partial evaluation. On the other hand, the action α_2 will be instantiated as `remove` in program slicing and `reduce` in partial evaluation.

The last stage, transformation, produces a residual program according to the decisions provided by Action Analysis. The specification for transformation handling slicing actions is depicted in Figure 10 in Appendix C.

4.4 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 [27]). Nevertheless, we can still cast backward slicing into the unified framework described above.

It is interesting to point out that both forward and backward slicing share *identical* action analysis and transformation stage. The only difference lies in their slicing analysis specification. Just as in the case of forward slicing, we continue to define those variables declared in (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 a set of variables, at each program point, which must be made residual in order to enable computation of the values of those residual variables at the end of the program. If we continue to define σ and δ as the computation of transient and residual information respectively, backward slicing analysis specifies that δ, instead of σ, ensures the congruent division of programs.

Theorem 3 *The backward slicing analysis defines a projection over a 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. It is only by lifting the analysis to the specification level do we discover their similarity.

5 Benefits of The Framework

The unified framework shows that techniques of partial evaluation and (forward) program slicing are intimately related, despite the striking semantic anomaly between the results produced by each of these techniques. This framework theoretically captures the essence of program slicing and partial evaluation. In this section, we show some implications of this unified framework.

5.1 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 binding-time analysis can automatically become candidates for improving forward slicing analysis. These include techniques for weak staticness [14], partially static data, to name just a few.

As backward slicing is cast into the unified framework, it shares the wide variety of analysis improvement provided by the existing binding-time analysis and forward slicing analysis. For instance, using the technique available for handing partially-static data (as available in traditional partial evaluation), we can obtain a version of backward slicing that handles partially-transient data; this is intimately related to the backward slicing technique proposed in [35].

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 [8] (i.e. for each execution path in the program, we associated a corresponding value w.r.t to the constraint) can be applied in partial evaluation.

5.2 Tuning Existing Specialization Techniques

The framework implies that it is possible to produce a semantic-based forward slicing process: One just needs to do the following:

1. Replace the usual `remove` action in slicing by the `reduce` action from partial evaluation; certainly, concrete static values have to be provided to the static program input during the transformation stage.

2. Modify the `retain` action at the transformation stage so that additional assignment statements are constructed before the statement to be retained. These assignment statements aim to residualize the static variables used at the retained statement. For example, given a statement `u := x - v` to be retained, with `x` being static and `u` and `v` being residual. Furthermore, let the value of `x` be `10` just before the action to retain the assignment is taken. With the modified `retain` action, the residual code will become (`x := 10; u := x - v`) instead of simply

(u :=x - v).

In effect, the derived semantic-based slice can be perceived as a residual program produced by a coarse partial evaluator in which the `residualize` action has been simplified.

5.3 Combining Partial Evaluation and Backward Slicing

While it is not sound in general to simply replace all the `reduce` actions in a partial evaluation process by `remove` actions, we can imagine a variant of partial evaluation in which *some* of the `reduce` actions are replaced by `remove` actions.

Specifically, for those static variables whose values do not contribute to the production of residual codes, their computation can be omitted (as signified by the `remove` action). In general, a separate analysis is needed to determine those static variables and static statements which do not contribution to the production, and it is questionable whether there will be much, if any, efficiency gain from this variant. Such a variant of partial evaluation can be easily obtained with the help of backward slicing.

We consider specialization of a program *wrt both* a set of static input variables *and* a backward slicing criterion, i.e. the specialization contexts deal with both *usability condition* and the *degree of knowledge* of the program variables[6].

With two sets of specialization contexts to be propagated in the reverse direction, we can perform two different residual analysis separately to obtain pairs of residual information for each of the variables: its binding-time value and its residual value. This information pair can be used to drive the action analysis specifically for this specialization. The following table describes their impacts:

	Static	*Dynamic*
Transient	Redundant	Redundant
Residual	`reduce`	`residualize`

Since a variable with transient value will not contribute to the construction of final residual program, a statement which comprises of only such variables can be safely removed from the residual program. On the other hand, variables with static and residual values will be *reduced* by its actual values during specialization, and variables with dynamic and residual values can be residualized.

Finally, from the perspective of specialization, a typical backward slicing requires minimum information from the specialization context; it simply classifies the variables in a specialization context as either transient or residual. We hope that with more specific specialization-context information, such as the constancy of some output (transient) variables, a backward specialization will produce a more refined specialized program. Some backward specialization, such as [35], has exploited static data construction at the output. For a general specialization context, we believe that the specialization must be ready to handle *constraint*. This leads us to the investigation of constraint-based specialization in future work.

6 Future Work

In previous sections, we briefly introduce existing partial-evaluation and slicing techniques. We also described in detail a unified framework that captures the essence of these two techniques and summarized the benefits of this framework. Next we outline the future direction and related research issues we aim to resolve.

6.1 Constraint-based Framework

There have been some works on constraint-based slicing and constraint-based partial evaluation, e.g. [9, 16, 8, 39, 10, 21, 28, 27]. A constraint-based partial evaluation/slicing is usually perceived as a generalization of typical partial evaluation/slicing. In these works, constraints are propagated throughout a program via symbolic predicate transformers. The primary objective of introducing constraints, in both partial evaluation and slicing, is to enable aggressive elimination of branches

of conditional statements. This is contrary to the *typical* specialization which we have described so far, in which the residual information of program variables is represented using two values.

In spite of the inclusion of constraints, a constraint-based specialization should still provide the usual classification of variables at program input and/or output into transient/static and residual/dynamic. As the framework reveals, such classification is necessary for the effective execution of Action Analysis. It is also beneficial to use constraints to enrich and refine the existing three-point domain of the offline residual analysis, in the typical partial evaluation/slicing transformation which involves constant inputs.

The unified framework provides several new perspectives to the technique of constraint-based slicing/partial evaluation:

1. *Constraints are projections.* We can view constraints as functions of type $V \rightarrow V$, specifying partitioning of program states. As such, constraints can be treated as projections, and constraint propagation defined by predicate transformers can be viewed as a projection-based analysis. Viewing constraints as projections over program states, our framework can also be used to capture the essence of constraint-based slicing [9] and constraint-based partial evaluation [28, 21, 27].

2. *offline-constraint based specialization.* The offline framework leads us to consider the possibility of defining projections/constraints, at the residual analysis stage, as an approximation to (or a weakening of) actual constraints provided by the specialization context. This opens up the opportunity to trade specialization efficiency for the degree of refinement of residual codes. We have not seen any existing constraint-based specialization techniques that make use of such trade-offs.

   In the case when constraints are provided only to transient/static variables, but not to residual/dynamic variables, the constraint-based residual analysis will involve mixed computation of constraints and "unknown" information, denoted by $\top$ (similar to $\overline{\natural}$ in residual analysis).

Mixed computation with the $\top$ element has been well-studied in the domain of abstract interpretation.

The offline-constraint based specialization here is quite closely related with *abstraction-based specialization* [18, 23, 26]. We will investigate these techniques in future.

3. *Constraint for residual/dynamic variables.* This is primarily used to provide more refined information for residual/dynamic variables. However, such a specification blurs the distinction between transient/static and residual/dynamic variables, the advantage of which is presently unclear.

6.2 Forward Analysis and Backward Analysis

Our unified framework demonstrates that, from a high-level point of view, both forward analysis (e.g. binding time analysis and forward slicing analysis) and backward analysis (e.g. backward slicing analysis) can be treated as projection-based analyses on well-classified information. However, algorithmically, backward analysis will be backward in nature and thus be distinct from forward analysis. Naturally we perform forward analysis and backward analysis separately and independently (in two phases) when we are supplied with a pair of input and output specialization contexts.

We have seen situation in which both input and output information are needed during specialization. From the correctness (of program transformation) point of view, we would like to know how these input and output information can be reconciled to ensure transformation correctness. More interesting, we plan to derive an efficient algorithm for analyzing program wrt both input and output information. We will investigate the relationship between forward analysis and backward analysis, figuring out the essence of these two analyses.

6.3 New Specialization Techniques and Implementation

In Section 5, we discuss the opportunities of cross-fertilization between partial evaluation and slicing. We hope to develop some slicing variants and/or PE variants by applying the existing techniques and theories to each other. One of the possible solutions is that we reformulate those variants (e.g. weak staticness, partially static data) by using projection and then apply these projection into the corresponding analysis phase.

Inter-procedural specialization plays important roles in real world. Specialization across procedures complicates the situation due to the necessity of translating and passing the specialization context into and out of calling and called procedures. There have been some research works on inter-procedural static slicing [40, 2, 20, 3, 19, 29]. These existing algorithms for computing inter-procedural static slice involved iterative data-flow computation ,or transitive traversal over system dependence graph which is an extension of program dependence graph by adding vertices and edges representing call statements, parameters passing. We will explore the opportunities of projection-based analysis in inter-procedural program slicing and function specialization to find a more efficient way to do inter-procedural Specialization.

We intend to investigate the extension of object language by adding some features, such as by adding function/procedure invocation and pointer. The framework does not include many different preprocessors which are commonly added to a partial evaluator, such as Tempo [13]. These preprocessors include alias analysis, side-effect analysis, use-sensitivity analysis, evaluation-time analysis, *etc.* In future we plan to investigate the effect of these analyses in the presence of both program slicing and partial evaluation.

We intend to implement a fully automated system that achieves the potential of the theories in this framework.

7 Conclusion

In this technical report, 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 programs. In our model both static/transient and dynamic/residual data co-existent. Based on the model, we demonstrate that the forward slicing analysis and binding-time analysis are both projection-based analyses of the same kind, while the backward slicing analysis is a projection-based analysis over residual data.

Interestingly, all the three transformations make the same decision about transformation actions, despite the fact that slicing chooses `remove` and `retain` actions, and partial evaluation chooses `reduce` and `residualize` action.

This framework captures the essence of program slicing and partial evaluation. Actually, many existing specialization techniques can be cast into this framework. The uniformity in transformation also makes it possible and meaningful to combine program slicing and partial evaluation into one phase.

Based on this unified framework, we enable cross-fertilization between program slicing and partial evaluation. We also demonstrate how existing forward slicing can be tuned into a semantic-based forward slicing and how to compose partial evaluation and backward slicing to form a new transformation. We also discuss the possibility of employing offline constraint-based transformation to trade between transformation efficiency and degree of refinement of specialized code.

References

[1] T. Amtoft. Partial evaluation for constraint-based program analyses. Technical Report BRICS-RS-99-45, Department of Computer Science, University of Aarhus, 1999.

[2] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Commun. ACM*, 21(9):724–736, 1978.

[3] J. Bergeretti and B. A. Carr. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985.

[4] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

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

[6] G. Bobeff and J. Noyé. Molding components using program specialization techniques. *In J. Bosch, C. Szyperski, and W. Weck, editors*, July 2003.

[7] G. Bobeff and J. Noyé. Component specialization. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 39–50. ACM Press, 2004.

[8] R. Hierons C. Fox, M. Harman and S. Danicic. Backward conditioning: A new program specialisation technique and its application to program comprehension. In *Proceedings of the Ninth International Workshop on Program Comprehension (IWPC'01)*, pages 89–97. IEEE Computer Society, 2001.

[9] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607, 1998.

[10] J. J. Comuzzi and J. M. Hart. Program slicing using weakest preconditions. In *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, pages 557–575. Springer-Verlag, 1996.

[11] C. Consel. A tour of schism: a partial evaluation system for higher-order applicative languages. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 145–154. ACM Press, 1993.

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

[13] C. Consel, J.L. Lawall, and A.-F. Le Meur. A tour of Tempo: A program specializer for the C language. *Science of Computer Programming*, 2004.

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

[15] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[16] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. Consit: a fully automated conditioned program slicer. *Software Practice Experience*, 34(1):15–46, 2004.

[17] M. Harman and R. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[18] J. Hatcliff, M. B. Dwyer, and S. Laubach. Staging static analyses using abstraction-based program specialization. In *Proceedings of the 10th International Symposium on Principles of Declarative Programming*, pages 134–151. Springer-Verlag, 1998.

[19] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[20] M. Du J. Hwang and C. Chou. Finding program slices for recursive procedures. In *In Proceedings of the Twelfth International Computer Software and Applications Conference (COMPSAC 88)*, pages 220–227. ACM Press, October 1988.

[21] Y. Jin and C.Z Jin. Constraint-based partial evaluation for imperative languages. *Journal of Computer Science and Technology*, 17(1):64–72, 2002.

[22] N. D. Jones. Mix: Ten years after. In *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM Press, 1995.

[23] N. D. Jones. The essence of program transformation by partial evaluation and driving. In *Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 62–79. Springer-Verlag, 2000.

[24] N.D. Jones. Automatic program specialization: A re-examination from basic principles. In *In [5]*, pages 225–282, 1988.

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

[26] S. C. Khoo. *Parameterized partial evaluation principle and practice*. PhD thesis, Department of Computer Science, Yale University, 1993.

[27] S. C. Khoo and K. Shi. Program adaptation via output constraint specialization. *Journal of Higher-Order and Symbolic Computing (HOSC)*, 17 (1-2):93–128, March - June 2004.

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

[29] A. Lakhotia. Improved interprocedural slicing algorithm, 1992.

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

[31] J. Launchbury. Strictness and binding-time analyses: two for the price of one. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 80–91. ACM Press, 1991.

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

[33] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, 1984.

[34] T. Reps and W. Yang. The semantics of program slicing. Technical Report TR-777, Computer Science Dept., Univ. of Wisconsin, June 1988.

[35] T. W. Reps and T. Turnidge. Program specialization via program slicing. In *Selected Papers from the Internaltional Seminar on Partial Evaluation*, pages 409–429. Springer-Verlag, 1996.

[36] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.

[37] F. Tip. A survey of program slicing techniques. Technical report, CWI (Centre for Mathematics and Computer Science), 1994.

[38] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 107–119. ACM Press, 1991.

[39] G. S. Yoon W. K. Lee, I. S. Chung and Y. R. Kwon. Specification-based program slicing and its applications. *J. Syst. Archit.*, 47(5):427–443, 2001.

[40] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[41] M. D. Weiser. *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.

Appendix A: Specification for Residual Analysis

The *Residual analysis* function $\mathcal{R}$ takes a program (Pgm) and an abstract program state ($AbsPgmState$), and returns a program associated with its updated abstract program state ($CtxPgm$).

The VAR function takes a sequence of statements and returns a set of variables appearing in these statements.

$$
\begin{array}{l}
\textit{Residual analysis } \mathcal{R} : Pgm \rightarrow AbsPgmState \rightarrow CtxPgm \\
\\
\mathcal{R}[\![S_1;\ S_2]\!]\theta \ = \ let\ [\![S_1]\!]^{\theta'} = \mathcal{R}[\![S_1]\!]\theta\ in\ \mathcal{R}[\![S_2]\!]\theta' \\
\mathcal{R}[\![Var := e]\!]\theta \ = \ [\![Var := e]\!]^{\theta[Var \leftarrow \hat{\mathcal{E}}[\![e]\!]\theta]} \\
\mathcal{R}[\![\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2]\!]\theta \ = \\
\quad if\ \hat{\mathcal{E}}[\![e]\!]\theta = \bar{\natural} \\
\quad then\ let\ \theta' = \theta[var_i \leftarrow \bar{\natural}]\ where\ var_i \in VAR(S_1, S_2) \\
\qquad in\ [\texttt{if } e^{\theta} \texttt{ then } S_1^{\theta'} \texttt{ else } S_2^{\theta'}]^{\theta'} \\
\quad else\ let\ [\![S_1]\!]^{\theta_1} = \mathcal{R}[\![S_1]\!]\theta \\
\qquad [\![S_2]\!]^{\theta_2} = \mathcal{R}[\![S_2]\!]\theta \\
\qquad \theta_3 = \theta_1 \sqcup \theta_2 \\
\qquad in [\![\texttt{if } e^{\theta} \texttt{ then } S_1^{\theta_1} \texttt{ else } S_2^{\theta_2}]\!]^{\theta_3} \\
\mathcal{R}[\![\texttt{while } e\ S]\!]\theta \ = \\
\quad if\ \hat{\mathcal{E}}[\![e]\!]\theta = \bar{\natural} \\
\quad then\ let\ \theta' = \theta[var_i \leftarrow \bar{\natural}]\ where\ var_i \in VAR(S) \\
\qquad in [\![\texttt{while } e^{\theta}\ S^{\theta'}]\!]^{\theta'} \\
\quad else\ let [\![S]\!]^{\theta'} = \mathcal{R}[\![S]\!]\theta \\
\qquad in\ [\![\texttt{while } e^{\theta}\ S^{\theta'}]\!]^{\theta\prime}
\end{array}
$$

Figure 8: Specification for Residual Analysis

Description of *Specification for Residual Analysis*:

1. *Sequences*: This specification shows that in this analysis the context information is propagated forwardly, i.e. the state of a node will affect its successive nodes.

2. *Assignments*: According to the definition of the *Abstract Evaluation Function* $\hat{\mathcal{E}}$ presented in Figure 7, if the abstract value of the rhs expression is $\bar{\natural}$, the abstract value of some variables in this rhs expression must be $\bar{\natural}$, i.e. these variables are classified as *residual*. Thus according

to the definition of forward slicing analysis or binding-time analysis, the lhs variable which is affected by the *use* of the variables appearing in the rhs expression, will also be classified as *residual*. In this way we capture the data dependency between the *definition* of lhs variable and the *use* of the variables appearing in the rhs expression which are defined in previous nodes.

3. *Conditions and Loops* : If the abstract value of the condition test expression is $\overline{\natural}$, the abstract value of some variables in this condition test expression must be $\overline{\natural}$, i.e. these variables are classified as *residual* . Because all the statements in the two branches of the `Condition` statement (or in the loop body of the `Loop` statement) are control dependent on this condition test node, we set the abstract value of all the variables appearing in the two branches of the *Condition statement* (or in the loop body of the *Loop statement*) as $\overline{\natural}$. If the abstract value of the condition test expression is $\underline{\natural}$, we just step into the two branches respectively and update their associated contexts based on the abstract context of their previous node. In this way we capture the control dependency between the condition test node and the statements in the two branches of *Condition statement* (or in the loop body of the *Loop statement*).

Appendix B: Specification for Action Analysis

The *Action analysis* function $\mathcal{A}$ takes a $CtxPrm$ and returns a program $AnnPgm$ tagged with annotation values $\{\alpha_1, \alpha_2\}$.

The action α_1 will be instantiated as `retain` in program slicing and `residualize` in partial evaluation. On the other hand, the action α_2 will be instantiated as `remove` in program slicing and `reduce` in partial evaluation..

Action analysis $\mathcal{A} : CtxPgm \rightarrow AnnPgm$

$\mathcal{A}[\![Var := e]\!]^{\theta} \ =$
$\quad if\ \theta|_{var} = \overline{\natural}$
$\quad then\ [\![Var := e]\!]^{\alpha_1}$
$\quad else\ [\![Var := e]\!]^{\alpha_2}$
$\mathcal{A}[\![\texttt{if}\ e^{\theta_1}\ \texttt{then}\ S_1^{\theta_2}\ \texttt{else}\ S_2^{\theta_3}]\!]^{\theta_4} \ =$
$\quad if\ \alpha[\![e]\!]\theta_1 \ = \ \overline{\natural}$
$\quad then\ [\![\texttt{if}\ e^{\alpha_1}\ \texttt{then}\ S_1^{\alpha_1}\ \texttt{else}\ S_2^{\alpha_1}]\!]$
$\quad else\ [\![\texttt{if}\ e^{\alpha_2}\ \texttt{then}\ \mathcal{A}[S_1]^{\theta_2}\ \texttt{else}\ \mathcal{A}[S_2]^{\theta_3}]\!]$
$\mathcal{A}[\![\texttt{while}\ e^{\theta_1}\ S^{\theta_2}]\!]^{\theta_2} \ =$
$\quad if\ \alpha[\![e]\!]\theta \ = \ \overline{\natural}$
$\quad then\ [\![\texttt{while}\ e^{\alpha_1}\ S^{\alpha_1}]\!]$
$\quad else\ [\![\texttt{while}\ e^{\alpha_2}\ \mathcal{A}[S]\theta_2]\!]$
$\mathcal{A}[\![S_1^{\theta_1};\ S_2^{\theta_2}]\!]^{\theta_2} \ = \ \mathcal{A}[\![S_1]\!]^{\theta_1}; \mathcal{A}[\![S_2]\!]^{\theta_2}$

Figure 9: Specification for action analysis

Appendix C: Specification for Transformation

The $Transformation$ function $\mathcal{T}$ takes an annotated program $AnnPgm$ and a specialization environment $SpeEnv$, return a specialized program (Pgm).

The specialization environment $SpeEnv$ play no role in program slicing while it will supply the concrete values for partial evaluation. In the follow figure, we demonstrate the specification for transformation in slicing, where a statement will be replaced by `skip` statement or retain in final specialized program.

$Transformation\ \mathcal{T} : AnnPgm \to SpeEnv \to Pgm$

$\mathcal{T}[\![Var := e]\!]^{Ann}\eta\ =$
$\quad if\ Ann = RET$
$\quad then\ [\![Var := e]\!]$
$\quad else\ [\![\texttt{skip}]\!]$
$\mathcal{T}[\![\texttt{if}\ e^{Ann_1}\ \texttt{then}\ S_1^{Ann_2}\ \texttt{else}\ S_2^{Ann_3}]\!]\eta\ =$
$\quad if\ Ann_1 = RET$
$\quad then\ [\![\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]$
$\quad else\ let\ Pgm_1 = \mathcal{T}[S_1]^{Ann_2}\eta$
$\qquad\qquad Pgm_2 = \mathcal{T}[S_2]^{Ann_2}\eta$
$\qquad in\ [\![\texttt{if skip then}\ Pgm_1\ \texttt{else}\ Pgm_2]\!]$
$\mathcal{T}[\![\texttt{while}\ e^{Ann_1}\ S^{Ann_2}]\!]\eta\ =$
$\quad if\ Ann_1 = RET$
$\quad then\ [\![\texttt{while}\ e\ S]\!]$
$\quad else\ [\![\texttt{while skip}\ \mathcal{T}[\![S]\!]^{Ann_2}\eta]\!]$
$\mathcal{T}[\![S_1^{Ann_1};\ S_2^{Ann_2}]\!]\eta\ =\ \mathcal{T}[\![S_1]\!]^{Ann_1}\eta; \mathcal{T}[\![S_2]\!]^{Ann_2}\eta$

Figure 10: Specification for transformation in slicing