DOI 10.1007/s00165-005-0077-6
BCS © 2006
Formal Aspects of Computing (2006) 18: 103–119

Formal Aspects of Computing

A formal relationship between program slicing and partial evaluation

David W. Binkley[1], Sebastian Danicic[2], Mark Harman[3], John Howroyd[4] and Lahcen Ouarbya[2]

[1] Department of Computer Science, Loyola College, 4501 N. Charles Street, Baltimore, Maryland 21210-2699, USA
[2]Department of Computing, Goldsmiths College, University of London, New Cross, London SE14 6NW, UK
[3]Department of Computer Science, King's College London, Strand, London WC2R 2LS, UK
[4]@UK PLC, 5 Jupiter House, Calleva Park, Aldermaston, Berkshire, RG7 8NN, UK

Abstract. A formal relationship between program slicing and partial evaluation is established. It is proved that for terminating programs, a residual program produced by partial evaluation is semantically equivalent to a conditioned slice.

Keywords: Conditioned program slicing; Partial evaluation

1. Introduction

Partial evaluation and slicing are both program specialisation techniques that attempt to simplify a program. Partial evaluation [JGS93] involves the evaluation of static computation, while slicing [BH04, Tip95] involves the removal of parts of the program which do not affect the values of selected variables at execution points of interest.

Both techniques have common applications including optimisation, program comprehension, dependence analysis and testing [BF98, CHM+98a, Das98, DFM96, DHN98, Ers78, JR94, HD97]. It may be productive, in the future, therefore, to explore the similarities and differences between partial evaluation and slicing and to investigate hybrid approaches that combine the two. In spite of this, research on partial evaluation and slicing has largely been conducted by two disjoint communities and so there has been relatively little work on the similarities and differences between the two techniques and none which establishes any formal relationship between the two.

This paper explores these similarities and differences, establishing a formal relationship between the residual program of partial evaluation and a form of slice known as a conditioned slice [CCD98, DFHH00, DDH+02, DDH04, FDHH04, DDF+04, HHD+01, FHHD01, HHF+02]. The paper uses the program projection framework [HBD03, HD97] which allows for the definition of various forms of program projection, which preserve only a projection of the original program's semantics.

The rest of this paper is organised as follows: Sects. 2 and 3 briefly review partial evaluation and slicing respectively. A semantics that is preserved by slicing is given in Sect. 4. Section 5 introduces the theoretical framework of program projection which allows different transformation methodologies to be compared. It is shown how different slicing techniques can be expressed using the framework of Sect. 5 and the semantics of Sect. 4. Section 6 expresses partial evaluation in terms of a framework in order to formally compare it with slicing. It is

Correspondence and offprint requests to: M. Harman, Software Engineering Group, Department of Computer Science, King's College London, Strand, London WC2R 2LS, UK. E-mail: Mark@dcs.kcl.ac.uk

	x	n	r
`r=1;`	D	S	S
`while (n>0)`	D	S	D
`{ n=n-1;`	D	S	D
`r=r*x;}`	D	S	D
`printf("%d",r);`	D	S	D

Fig. 1. Computing the power

established that a residual program of a terminating program is semantically equivalent to a conditioned slice. Section 7 describes some applications of the theoretical results of the paper, showing how partial evaluation and amorphous slicing can be combined to produce amorphous conditioned slices and how the precision of syntax preserving conditioned slicing could be improved using information from partial evaluation. Section 8 presents related work and Section 9 summarises the paper.

2. Partial evaluation

Many programs contain computations that can be performed at compile time. Optimising compilers exploit such computations by pre-computing their values. Partial evaluation (also known as mixed computation) takes this process one step further by allowing some of the inputs to a program to be statically determined (or statically 'bound'). By propagating that binding-time information 'forward' through the program, additional static computations can be identified. Static computations can be performed at compile time, often reducing the program's size and run time.

Partial evaluation was originally considered as an extension of compiler theory, used as a means of improving efficiency [Ers78] and of creating compilers and compiler generators from interpreters [Fut71, Fut99a, Fut99b, JSS89]. Initial work concerned declarative languages, for which the semantics were comparatively 'clean' [BHO$^+$76, GJ89, JSS85, Har78, Ste75]. Consequently, the propagation of binding time information was definable in terms of the formal semantics of the programming language concerned. More recently, partial evaluation-based approaches to software engineering [CHM$^+$98a], program comprehension [BF98], state-based testing [DHN98] and multimedia computing [Dra98] have been explored. The initial work on declarative language partial evaluation has also been extended to the imperative programming paradigm, most notably by the early work on [And92, GJ91, KKZG95] and by more recent work on partial evaluation using dependence graphs by Das [Das98] and the Tempo partial evaluation system for a large subset of the `C` programming language [CHM$^+$98b].

Algorithms for partial evaluation typically consist of two phases: an initial Binding Time Analysis (BTA) followed by a manipulation phase (called 'specialisation'). The result of this specialisation is a program, called the *residual program.* Some authors [RT96, SGJ94] use the term 'specialisation' to describe the more general concept of tailoring a program to a specific task. This nomenclature includes both partial evaluation *and* slicing as special cases of specialisation.

Example Consider[1] the `power` program in Fig. 1. Suppose the value of `n` is determined to be 4. The results of BTA for this program are shown in the right-hand sections of the figure (S indicates that a variable is determined to be static at the associated point, D indicates that it is determined to be dynamic). Using the results of BTA, it can be seen that the variable `n` remains static throughout the computation and therefore it will be safe to specialise the program by unfolding the loop controlled by `n`. This specialisation phase leads to the construction of the residual program in Fig. 2.

As originally formulated [BHO$^+$76, Ers78, Fut71, Har78, JSS85, Ste75], the residual program was constructed with respect to the instantiation of selected inputs with fixed values. Futamura and Nogi [FN87] introduced a more generalised form of partial evaluation, in which residual programs are constructed with respect to a set of initial states characterised by a first order predicate logic expression on input variables. Glück and Sørensen [GS96] cite four varieties of partial evaluator. These are:

[1] In order to simplify the following example, it will be assumed that the input to a program is taken from those variables that are referenced before being defined. This will allow program fragments and whole programs to be considered in a similar way.

```
r=1;
r=r*x;
r=r*x;
r=r*x;
r=r*x;
printf("%d",r);
```

Fig. 2. The residual power computation for `n` $= 4$

`base = 2;`		
`count = 0;`		
`for(i=0;i<N;i++)`	`for(i=0;i<N;i++)`	
`  { if (A[i] % base == 0)`	`  {`	
`      count++;`		
`    R[N-i-1] = A[i];`	`    R[N-i-1] = A[i];`	
`  }`	`  }`	
`b = (A[0]==R[0]) ;`	`b = (A[0]==R[0]) ;`	`b = (A[0]==A[N-1]);`
Original	Syntax-Preserving end-slice for `b`	Amorphous slice for `b`

Fig. 3. Amorphous and syntax-preserving slices

- Monovariant, where each function of the original program results in at most one function in the residual program.
- Polyvariant where each function of the original program may result in more than one function in the residual program.
- Monogenetic where each function in the residual program arises from exactly one function from the original.
- Polygenetic where each function in the residual program may arise from more than one function from the original.

These varieties only differ syntactically. Semantically they are all equivalent to generalised partial evaluation. This generalised form of partial evaluation is closely related to conditioned [CCD98, LCYK01, HHD$^+$01] and constrained [FRT95] slicing, as will be shown in Sect. 6.

3. Program slicing

Program slicing is a program simplification technique that removes from a program components (e.g., statements) that do not affect a computation of interest. The resulting (smaller) program, called a slice, captures a projection of the semantics of the original program. Figure 3 illustrates program slicing with a simple example of a program, p, and one of its slices, p'.

Slicing is useful in many aspects of software engineering because it preserves a projection of the semantics of the original program while reducing the size of the program. Originally slicing was applied to debugging [Wei79, Wei82, LW87] and (more recently) to algorithmic debugging [Kam93, Sha91]. Many other applications have also been investigated, including cohesion measurement [BO94, Lak93, OT93], re-engineering [LD98, LE93, SVM$^+$93], component re-use [BE93, CDLM96], program comprehension [DFM96, JR94, HD97], maintenance [GL91], testing [Bin98, Bin97, GHS92, HD95, HHD99, HHF$^+$02], program integration [HPR89] and software conformance certification [KS98]. There are several surveys of slicing, techniques and applications [BG96, HH01, De 01, Tip95].

In addition to capturing a projection of a program's semantics, the original application of slicing required a syntactic property: a slice must preserve a projection of the original program's syntax. Traditionally, slicing has used a 'statement deletion' for this projection largely for historical reasons. This derives from the original application of slicing [Wei79] to debugging, where the motivation for slicing was to remove statements and associated control structures that could not have been responsible for an error. The importance of this syntactic requirement varies from application to application. For re-engineering, program comprehension and testing it is primarily the semantic property of a slice that is of interest.

Conditioned slicing [CCD98] is the act of simplifying a program assuming that the states of the program at certain chosen points in its execution satisfy certain properties. What distinguishes conditioned slicing from other forms of slicing is mainly its implementation [DFHH00, DDH+02, FDHH04, DDF+04]. Conditioned slicing combines the standard techniques of slicing with symbolic execution and theorem proving. Conditioned slicers are required to reason about the validity of paths under certain conditions using symbolic execution and automated theorem proving. The simplifying power of the conditioned slicer depends on two things:

1. The precision of the symbolic executor which handles propagation of state and path information.
2. The precision of the underlying theorem prover which determines the truth of propositions about states and paths.

By using an approximation to a program's semantics using a form of symbolic execution, and by being willing to accept approximate results from the theorem proving itself, conditioning allows us to adopt reasoning that does not require the full force of inductive proofs. The theorem proving used in program conditioning is lightweight when compared to the theorem proving required for a complete formal analysis of a program. As in the case of traditional slicing, conditioned slicing only simplifies programs by deletion. Semantically, as will be seen later, conditioned slicing is very closely related to both traditional slicing and partial evaluation.

Amorphous slicing [HBD03] is a variation of slicing, in which simplification is not limited to statement deletion. For this reason it is clearly closer to partial evaluation than the other forms. To illustrate, consider the example in Fig. 3. The original program reverses the array `A`, keeping a count (in the variable `count`) of the number of array elements which are exactly divisible by `base`. The program also records whether or not the first and last element of the array are identical (in the variable `b`). The integer `N` (the size of the array `A`), is a compile-time constant. The array `A` is an input parameter to the program fragment. The original program is in the leftmost column of the figure. The syntax-preserving end-slice for the value of the variable `b` is in the central column and the amorphous slice is in the rightmost column of the figure.

Figure 3 makes it clear that amorphous slices can be smaller than their syntactically constrained counterparts. Of course, it is a more challenging problem to compute amorphous slices. Recently proposed algorithms [Bin99, HHZM01] represent initial attempts at addressing this problem.

Notice also that the variable `base` is statically determined, and so it could be 'partially evaluated away' in the original program and in its syntax-preserving end-slice. Since `N` is also statically determined, the loop can also be replaced by a sequence of `N` assignments. The combination of these partial evaluation steps together with syntax-preserving end-slicing, could almost produce the amorphous slice, as shown in Fig. 4. This suggests that amorphous slicing might be achievable as a combination of syntax-preserving end-slicing and partial evaluation. However, the program produced by mixing partial evaluation and syntax-preserving end-slicing, is a valid slice only because the computation was static for the unwanted array elements, `R[i]` for `(i<N)` and the unwanted variables `base` and `count`. Had the value of `N` or `base` been inputs, then partial evaluation would have had no effect. However, this example shows that there is a connection between the residual program of partial evaluation and amorphous forms of program slicing. This observation motivates the formal examination of this connection which follows.

4. Semantics of slicing

Originally, program slicing was defined in terms of an algorithm [Wei79]. Informally, this algorithm simplified programs both syntactically, and semantically. Given a program P and a set of variables J, Weiser's algorithm produces a program P' smaller than P which 'behaves the same' with respect to J. Weiser, in fact proved that in all states where P terminates, P' also terminates and agrees with P on all the variables in J.

Since Weiser, many other slicing algorithms have emerged. These algorithms are unified by the fact that they all attempt to simplify programs and at the same time preserve semantic information. They differ sometimes subtly, sometimes dramatically, in both the sort of simplification they perform and the kind of semantic information they preserve.

In order to formally categorise these related algorithms, a *projection framework* [HBD03] was introduced. This framework has been used to express similarities and differences between several different forms of slicing techniques including amorphous and syntax-preserving end-slicing [HD97].

`base = 2;`		
`count = 0;`		
`for(i=0;i<N;i++)`		
`  { if (A[i] % base == 0)`	`if (A[0] % 2 == 0)`	
`      count++;`	`  count=1;`	
`    R[N-i-1] = A[i];`	`  R[N-1] = A[0];`	
`  }`		
`b = (A[0]==R[0]) ;`	$\vdots$	
	`if (A[N-1] % 2 == 0)`	
	`  count++;`	
	`  R[0] = A[N-1];`	`R[0] = A[N-1];`
	`b = (A[0]==R[0]) ;`	`b = (A[0]==R[0]) ;`
Original	Partially Evaluated	Syntax-Preserving end-slice for `b`

Fig. 4. Amorphous slices computed by mixing partial evaluation and syntax-preserving slicing

$\mathcal{M} : P \to \Sigma_\perp \to \Sigma_\perp$

<u>skip statement</u>

$$\mathcal{M}[\![\text{skip}]\!]\sigma \triangleq \sigma$$

<u>*abort* statement</u>

$$\mathcal{M}[\![\mathit{abort}]\!]\sigma \triangleq \perp$$

<u>Assignment statements</u>

$$\mathcal{M}[\![x{=}e]\!]\sigma \triangleq \lambda\sigma[x \leftarrow \mathcal{E}[\![e]\!]\sigma]$$

(where the notation $\sigma[x \leftarrow e]$ represents the function which is the same as σ except that x gets mapped to e.)

<u>Sequences of statements</u>

$$\mathcal{M}[\![S_1; S_2]\!]\sigma \triangleq \mathcal{M}[\![S_2]\!](\mathcal{M}[\![S_1]\!]\sigma)$$

<u>`if` statements</u>

$$\mathcal{M}[\![\text{if } (b)\ S_1 \text{else } S_2]\!]\sigma \triangleq \mathcal{E}[\![b]\!]\sigma \to \mathcal{M}[\![S_1]\!]\sigma, \mathcal{M}[\![S_2]\!]\sigma$$

(where the notation $e \to b, c$ is the expression which yields b if e is True, c if e is False and $\perp$ if e is $\perp$).

<u>`while` loops</u> $\mathcal{M}[\![\ \texttt{while}\ (b)\ S]\!] \triangleq \mathit{fix}\,(\lambda f \cdot \lambda\sigma \cdot \mathcal{E}[\![b]\!]\sigma \to f(\mathcal{M}[\![S]\!]\sigma), \sigma)$

Fig. 5. Standard semantics for a simple procedural language

An important fact to notice in attempting to categorise slicing algorithms is that most conventional slicing techniques do not preserve standard program semantics.[2] In certain states the slice of a program may terminate where the original did not. In [DHHO04] a semantics, $\mathcal{M}_L$, is defined which they prove is preserved by Weiser's algorithm and other slicing algorithms based on data and control dependence. $\mathcal{M}_L$ is finer grained than standard semantics $\mathcal{M}$, in the sense that it agrees with $\mathcal{M}$ in all states which lead to termination. $\mathcal{M}_L$ will, however, distinguish between states which are all considered $\perp$ in standard semantics. $\mathcal{M}_L$ is neither stronger, nor weaker than standard semantics i.e there are programs which are equivalent under standard semantics which are not equivalent under $\mathcal{M}_L$ and vice-versa. It has been proved [Oua05] that $\mathcal{M}_L$, unlike $\mathcal{M}$, (Fig. 5), has the property that it is preserved by slices produced by some conventional slicing algorithms.

[2] Other authors have produced semantics which they claim to be preserved by slicing. These include the lazy semantics of Cartwright and Felleisen [CF89] and the transfinite semantics of Giacobazzi and Mastroeni [GM03].

In $\mathcal{M}_L$, as in the the lazy semantics of Cartwright and Felleisen [CF89] states can be partially defined. This means that some variables may be mapped to $\perp$ and others to well defined values. The set of such states is denoted as $\Sigma^{\perp}$.

$$\Sigma^{\perp}: I \to V_{\perp} \qquad \text{where } V_{\perp} \triangleq V \cup \{\perp\}$$

and I is the set of all variables and V the set of all possible values that can be assigned to these variables. The ordering, $\sqsubseteq$, on $\Sigma^{\perp}$ is a much richer ordering than the ordering on $\Sigma_{\perp}$ of standard semantics where all distinct non $\perp$ states are incomparable. For elements of $\Sigma^{\perp}$,

$$\sigma_1 \sqsubseteq \sigma_2 \quad \triangleq \quad \text{for all variables } x,\ \sigma_1(x) = \sigma_2(x) \ \text{ or } \ \sigma_1(x) = \perp.$$

Since variables can be mapped to $\perp$, there is the possibility that evaluating an expression in a partially defined state can yield $\perp$. Not all variables referenced by an expression necessarily contribute its value, for example, the value of the expression $x - x$ is independent of the value of x. For this reason, a function, *det* is defined which takes an expression e and returns the set of variables which really contribute to its value.

Definition 1 ($det : E \to P(V)$) Given an expression e, $det(e)$ is defined as follows:

1. For all $x \in det(e)$, there exists two states σ_1, σ_2 differing only on x such that: $\mathcal{E}[\![e]\!]\sigma_1 \neq \mathcal{E}[\![e]\!]\sigma_2$.
2. For all $x \notin det(e)$, and for all states σ_1, σ_2 differing only on x, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2$.

If $det(e)$ contains a variable which has $\perp$ as a value in σ, then the whole expression is evaluated to $\perp$ in σ. Otherwise the lazy value, $\mathcal{E}_L$, of an expression is the same as its strict value, $\mathcal{E}$. The meaning of an expression in our lazy semantics is the function, $\mathcal{E}_L: E \to \Sigma^{\perp} \longmapsto V_{\perp}$.

$$\text{given by } \mathcal{E}_L e\sigma \quad \triangleq \quad \begin{cases} \perp \text{ if } \exists v \in det(e) \text{ with } \sigma v = \perp. \\ \mathcal{E}e\sigma \text{ otherwise.} \end{cases}$$

The lazy meaning of a program is given by the function $\mathcal{M}_L$, which, as in the case of standard semantics, is a state to state function:

$$\mathcal{M}_L: P \longrightarrow \Sigma^{\perp} \longrightarrow \Sigma^{\perp}.$$

Lazy semantics of the `skip` statement

$$\mathcal{M}_L[\![\texttt{skip}]\!]\sigma \quad \triangleq \quad \sigma$$

As in standard semantics, the meaning of `skip` is the identity function on states.

Lazy semantics of the *abort* statement

$$\mathcal{M}_L[\![abort]\!]\sigma \quad \triangleq \quad \sigma$$

Unlike in standard semantics, the lazy meaning of the *abort* statement is the same as lazy meaning of the `skip` statement. This is a fundamental difference between lazy and standard semantics.

Lazy semantics of assignment statements

$$\mathcal{M}_L[\![\texttt{x=e}]\!]\sigma \quad \triangleq \quad \sigma[\texttt{x} \leftarrow \mathcal{E}_L[\![\texttt{e}]\!]\sigma]$$

As in standard semantics, the meaning of an assignment is obtained by updating the state with the new value of the variable assigned to it. In the case of lazy semantics, this value is the lazy value of the corresponding expression. Since in lazy semantics, there are states which map some variables to proper values and other variables to $\perp$, the assignment rule implies that a variable can 'recover' from being undefined as demonstrated by program p in Fig. 6, where after the loop `x` has the value $\perp$ but it recovers to 5 after the assignment `x=5`.

Lazy semantics of the statement sequences

$$\mathcal{M}_L[\![S_1\,;S_2]\!] \quad \triangleq \quad \mathcal{M}_L[\![S_2]\!] \circ \mathcal{M}_L[\![S_1]\!]$$

As in standard semantics, the lazy meaning of a sequence of statements is simply the composition of the meanings of the individual statements.

`x=1;`
`while (x>0)x=x+1;`
`x=5;`
	`x=1;`
	`if (z>0)`
	`{`
	`  x=1;`
	`  y=2;`
	`}`
		`while (true)`
		`{`
		`  x=1;`
		`  y=y+1;`
		`}`
			`x=1;`
			`z=1;`
			`while (true)`
			` if(y>0) y=-1;`
			` else  y=1;`
			`while (y>0)`
			`{`
			` x=1;`
			` z=2;`
			`}`

Program p	Program q	Program r	Program s

Fig. 6. Example programs

Lazy semantics of `if` **statements**

$$\mathcal{M}_L[\![\texttt{if}\,(B)\ S_1\ \text{else}\ S_2]\!]\sigma \quad \triangleq \quad \mathcal{E}_L[\![B]\!]\sigma \rightarrow \mathcal{M}_L[\![S_1]\!]\sigma, \mathcal{M}_L[\![S_2]\!]\sigma, \mathcal{M}_L[\![S_1]\!]\sigma \sqcap \mathcal{M}_L[\![S_2]\!]\sigma$$

where $\sigma_1 \sqcap \sigma_2$, the *meet* of σ_1 and σ_2, is defined as $\lambda i \cdot \sigma_1(i) = \sigma_2(i) \rightarrow \sigma_1(i), \perp$ and the notation $a \rightarrow b, c, d$ is shorthand for if a is true return b otherwise if a is false return c otherwise if a is $\perp$ return d and the notation $a \rightarrow b, c$ is shorthand for if a is true return b otherwise return c.

The difference between this and standard semantics is when the guard evaluates to $\perp$. If a variable x is assigned different values in the then and else parts its lazy value will be $\perp$. On the other hand, if the value of x is the same in the then and else parts then this should be its final value even if the guard is $\perp$, as, in this case, the value of x does not depend on the guard.

Consider, for example, program q in Fig. 6. Given an initial state $\sigma = \{x \mapsto 1, y \mapsto 1, z \mapsto \perp\}$, the value of the if predicate, $z > 0$ in σ is $\perp$. However, the final value of the variable x is 1 whichever branch of the conditional is executed. For this reason, the lazy value of the variable x after executing program q in σ is deemed to be 1.

The value of the variable y has different values after executing the then branch from when executing the else branch, and hence, the final value of the variable y is $\perp$.[3]

Before the lazy meaning of while loops can be given, *loop unfoldings*, $\mathcal{W}_n(B, S)$ for each $n \in \mathbb{N}$ are defined:

$$\begin{aligned} \mathcal{W}_0(B, S) &\triangleq \textit{abort} \\ \mathcal{W}_{n+1}(B, S) &\triangleq \texttt{if}\,(B)\ \{S; \mathcal{W}_n(B, S)\}\ \text{else skip} \end{aligned}$$

Lazy semantics of `while` **loops**

$$\mathcal{M}_L[\![\texttt{while}\,(B)\ S]\!] \quad \triangleq \quad \lambda\sigma \cdot \bigsqcup_{i=0}^{\infty}(G_i\sigma)$$

$$\text{where } G_i\sigma \quad \triangleq \quad \bigsqcap_{n=i}^{\infty} \mathcal{M}_L[\![\mathcal{W}_n(B, S)]\!]\sigma.$$

The *final lazy value* of variable x after executing a while loop starting in state σ is defined to be the limit of all the values of x after executing each of the unfoldings. If the limit does not exist, then the final lazy value is defined to be $\perp$. Here we mean the limit with respect to a discrete metric i.e. for the limit to exist, there must exist an $N \in \mathbb{N}$ such that all unfoldings greater than N give the same value for x in σ. If this is the case we say the value of x *stabilises* after N unfoldings. The lazy meaning of the while loop is thus the limit of the meet of the lazy meaning of all its corresponding unfoldings. Although the $\mathcal{M}_L[\![\mathcal{W}_n(B, S)]\!]$ is not monotonic, clearly $G_i \sqsubseteq G_{i+1}$, hence the least upper bound of the G_i exists.

[3] Unlike this semantics, the value of the variable x after executing program q in Fig. 6 in the state σ when using the lazy semantics by Cartwright and Felleisen [CF89] or the transfinite semantics by Giacobazzi and Mastroeni [GM03] is $\perp$. This is a result of the fact that their semantics loses all information about the variables defined in the then or else parts of an if statement in states where its predicate is undefined.

In states where the while loop does not terminate, if the value of the variable stabilises after i unfoldings for some $i \geqslant 0$, then its meaning will be the stabilised value. Otherwise, its value is just $\perp$, for example, given the infinite loop in program r in Fig. 6 the value of the variable x stabilises to 1 after the first unfolding whereas the value of the variable y never stabilises. In this case, the lazy values of x and y are 1 and $\perp$, respectively.

In states where the predicate of the while loop always evaluates to $\perp$, if the value of a variable is the same for all the unfoldings, this is its final lazy value and otherwise the variable is evaluated to $\perp$.

Consider, for example, program s in Fig. 6. After executing the first infinite loop in the value of the variable y is undefined and therefore, the condition of the second while loop is undefined. However, the value of the variable x does not change after executing the body of the loop zero, a finite or infinite number of times. In this case the final lazy value of x is equal 1.

The variable z, on the other hand, unlike the variable x, has a different value when the body of the while loop is not executed at all, from its value when the body is executed so in this case the final lazy value of z is $\perp$.

5. The program projection framework

The projection framework [HBD03, HD97] is defined with respect to two relations on programs: an ordering relation $\lesssim$ and an equivalence relation, $\approx$. The ordering, $\lesssim$, is normally used to capture the syntactic property which transformation systems such as slicing seek to optimise. Programs which are lower according to $\lesssim$ are considered to be better. The equivalence relation, $\approx$, captures semantic properties that must remain invariant and will normally be defined in terms of program semantics.

Definition 2 (($\lesssim, \approx$)-Projection) Given a symmetric transitive relation $\lesssim$, and an equivalence relation $\approx$, and programs p and q,

$$p \text{ is a } (\lesssim, \approx)\text{-}\textit{projection} \text{ of } q \quad \Longleftrightarrow \quad p \lesssim q \ \wedge \ p \approx q.$$

That is, the syntax must not 'get worse' as a result of projection of a program, while the semantics of the projected program must remain equivalent to the original.

Definition 3 (A program transformation satisfies a projection) A program transformation f satisfies a $(\lesssim, \approx)$-*projection* if and only if for all programs p, $f(p)$ is a $(\lesssim, \approx)$-*projection* of p.

Clearly, every program transformation satisfies the *weakest* $(\lesssim_A, \approx_A)$-projection where $p \lesssim_A q$ and $p \approx_A q$ for all programs p and q. Trivially, if a technique satisfies a particular $(\lesssim, \approx)$-projection then it satisfies all weaker projections. The weakest projection is useless as it tells us nothing. Clearly the stronger projection that we find, the more precisely we capture behaviour. The purpose of the projection framework is to capture the behaviour of each technique or algorithm sufficiently strongly to at least distinguish it from other techniques.

Alternatively, the projection framework can be used in the inverse manner for not categorising but defining classes of algorithms. We may define, say, a ζ-transformation to be any transformation which satisfies $(\lesssim_\zeta, \approx_\zeta)$ -projection. where $\lesssim_\zeta$ is a given ordering and $\approx_\zeta$ is a given equivalence as defined above.

5.1. Using the framework to express slicing

The projection framework has successfully been used to categorise a variety of slicing techniques. As an introduction to the projection framework it is shown how three kinds of slices: (traditional) syntax-preserving slices, amorphous slices, and conditioned slices are categorised using it. This subsection describes the syntactic and semantic relations for these three kinds of slices. It is the shown how partial evaluation can also be expressed using the framework.

5.1.1. Syntax-preserving slicing

It is helpful in discussing slicing to think of each line of a program as a single (labelled or numbered) node of a program's control flow graph (CFG).

The following definition formalises the oft-quoted remark:

"*a slice is a subset of the program from which it is constructed.*"

It defines *syntax-preserving ordering*, the ordering for syntax-preserving slicing, denoted $\lesssim_{SP}$. (In the sequel, subscripts are added to $\lesssim$ and $\approx$ to distinguish different instantiations of the respective relations.)

Definition 4 (Syntax-preserving ordering) Let F be a partial function which takes a program and returns a function from line numbers to statements, such that the function $F(p)$ maps l to c if and only if program p contains the statement c at line number l. Syntax-preserving syntactic ordering is defined as follows:

$$p \lesssim_{SP} q \Leftrightarrow F(p) \subseteq F(q).$$

Definition 5 (End-slice equivalence) Given two programs p and q, and a set of variables V, p is *end-slice semantically equivalent* to q, written $p \approx^{V}_{SP} q$, if and only if for all states σ, for all variables $v \in V$,

$$\mathcal{M}_L[\![p]\!]\sigma v = \mathcal{M}_L[\![q]\!]\sigma v.$$

Instantiating Definitions 4 and 5 into Definition 2, the following definition of an end-slice is obtained.

Definition 6 (Syntax-preserving end-slice) Program q is a *syntax-preserving end-slice* of program p with respect to set of variables V if and only if q is a $(\lesssim_{SP}, \approx^{V}_{SP})$-projection of p.

5.1.2. Amorphous slicing

To define amorphous slicing, only the syntactic relation needs to be altered; the semantic relation remains the same. For amorphous slicing, a natural choice for a syntactic relation is that p is less than q if it contains fewer statements. Such a definition of the syntactic relation naturally depends upon how statements are to be counted. The approach adopted in this paper, in keeping with the spirit of slicing, is to count the nodes in the CFG [HRB88, OO84].

Definition 7 (Amorphous syntactic ordering) Let $F(p)$ be the number of nodes in the CFG of program p. The amorphous syntactic relation, $\lesssim_{AS}$, is defined as follows:

$$p \lesssim_{AS} q \Leftrightarrow F(p) \leqslant F(q)$$

Definition 8 (Amorphous semantic equivalence) $\approx_{AS} = \approx_{SP}$

Definition 9 (Amorphous static slicing) Program q is an *amorphous static slice* of program p with respect to the set of variables V if and only if q is a $(\lesssim_{AS}, \approx^{V}_{SP})$-projection of p.

Amorphous slicing subsumes (i.e. it is weaker than) syntax-preserving end-slicing because $\lesssim_{SP} \subseteq \lesssim_{AS}$ and $\approx_{AS} = \approx_{SP}$. As a consequence of this, any implementation of syntax-preserving slicing is also an implementation of amorphous slicing, albeit a rather sub-optimal implementation. Therefore, there will always be an amorphous slice which contains no more CFG nodes than the syntax-preserving end-slice constructed for the same set of variables. Of course, there will often be an amorphous slice which contains fewer nodes than the corresponding syntax-preserving end-slice. This makes amorphous slicing attractive when preservation of syntax is unimportant.

5.1.3. Conditioned slicing

Finally, conditional slicing is defined in the projection framework. As will be seen, the residual program produced by partial evaluation is most closely related to a conditioned slice [CCD98].

Traditionally [CCD98, DFHH00] conditioned slices have been constructed by statement deletion. Thus, the definition of the conditioned slice syntactic ordering is the same as the syntax-preserving ordering used for syntax-preserving end-slicing.

Definition 10 (Conditioned syntactic ordering) The conditioned syntactic relation, $\lesssim_{CS}$, is defined as follows:

$$p \lesssim_{CS} q \Leftrightarrow p \lesssim_{SP} q$$

For example, the conditioned program in Fig. 7 is lower than the original program in the ordering of $\lesssim_{CS}$, because the conditioned program contains a subset of the statements of the original.

The semantic equivalence preserved by conditioned slicing is defined as follows:

Definition 11 (Conditioned semantic equivalence) p' is conditioned equivalent to p with respect to (V, Σ), written $p \approx^{(V,\Sigma)}_{CS} p'$ if and only if

$$\forall \sigma \in \Sigma, \forall v \in V, \mathcal{M}_L[\![p]\!]\sigma v = \mathcal{M}_L[\![p']\!]\sigma v$$

`x = 0 ;`	`x = 0 ;`
`y = 1 ;`	`y = 1 ;`
`if (x==0)`	
`    if   (y==2)`	
`           n = 3;`	
`         else n = 4;`	`n = 4;`
`  else   n = 5;`	
`if (n==3)`	
`   a = 5;`	
`  else a = 7;`	`a = 7;`
Original program	Conditioned program

Fig. 7. Conditioning

That is, for a set of states Σ and variables V conditioned slicing preserves the values of variables in V when the program is executed in a state in Σ. For example, the conditioned program in Fig. 7 is conditioned equivalent to the original with respect to (x, y) for when x is the set of all program variables and y is the set of all possible states, since the conditioned program and the original produce identical answers for all variables in all states.

Combining these two definitions into the framework yields the following definition

Definition 12 (Conditioned slice) Program q is a *conditioned slice* of program p with respect to the slicing criterion (V, i) if and only if q is a $(\lesssim_{CS}, \approx^{(V,\Sigma)}_{CS})$-projection of p.

6. A formal relationship between program slicing and partial evaluation

This section casts partial evaluation into the projection framework. The syntactic ordering relation, denoted by $\lesssim_{PE}$, is not typically defined explicitly. In order to fit partial evaluation into the projection framework it is necessary to make this relation explicit. In the literature on partial evaluation, it is implicitly taken to order programs according to their execution speed. Note that naming this "Residualisation syntactic ordering" highlights a bias from the slicing terminology; there is really no requirement that the ordering be syntax based.

Definition 13 (Residualization syntactic ordering) $p \lesssim_{PE} q$ if and only if p requires less execution time than q.

Definition 14 (Static variables) Given a program p and a set of states Σ, there is a set of variables, $SV_{(p,\Sigma)}$ which are 'static' with respect to Σ. That is, these variables have the same values in every computation of p, which starts in an initial state drawn from Σ. More formally, $x \in SV_{(p,\Sigma)}$ if and only if there do not exist states σ_1 and σ_2 in Σ with

$$\bot \neq \mathcal{M}[\![p]\!]\sigma_1 x \neq \mathcal{M}[\![p]\!]\sigma_2 x \neq \bot.$$

Partial evaluation is defined with respect to a set of states, Σ, and produces a residual program. The residual program must behave identically to the original when executed in a state in Σ for all dynamic (non-static) variables. However it need not preserve the meaning of the original program for static variables, as these can be 'residualised' and their computation may disappear. For example, for the `power` program, the set of states is $\Sigma_{\texttt{power}}$, defined as $\Sigma_{\texttt{power}} = \{\sigma \mid \sigma[\![\texttt{n}]\!] = 4\}$. The set of static variables for the `power` example, $SV_{(\texttt{power},\Sigma_{\texttt{power}})}$ is $\{\texttt{n}\}$, because in all terminating computations, the final value of `n` is zero, whereas for all other variables, there will exist (at least) two states in $\Sigma_{\texttt{power}}$ which produce different final values of `n`.

Definition 15 (Residualization semantic equivalence $\approx^{\Sigma}_{PE}$) Let I be the set of all program variables and let p and p' be programs. p is residually semantically equivalent to p' with respect to Σ if and only if $\forall \sigma \in \Sigma, \forall v \in (I - SV_{(p,\Sigma)}), \mathcal{M}[\![p]\!]\sigma v = \mathcal{M}[\![p']\!]\sigma v$.

Using $\approx^{\Sigma}_{PE}$ and $\lesssim_{PE}$ it is now possible to formulate residualisation within the projection framework.

Definition 16 (Residualisation) p' is a residualisation of p with respect to a set of states Σ if and only if it is a $(\lesssim_{PE}, \approx^{\Sigma}_{PE})$-projection of p.

The connections between theoretical and algorithmic aspects of partial evaluation and program slicing are formally established in this section. A key semantic difference between slicing and partial evaluation concerns the form of semantics preserved by each. Residualisation semantic equivalence (for C-like programs which are defined to have strict semantics) is, unsurprisingly, defined with respect to a strict semantics.

However, as was discussed in Sect. 4, slicing preserves lazy and not strict semantics. The nature of the semantics preserved by slicing and residualisation is therefore one of the key differences between the two approaches to program specialisation.

The syntactic ordering used in partial evaluation also differs from that used in both syntax-preserving and amorphous slicing. There are situations where the two relations will agree, for example, both would allow[4] the transformation

$[\![\texttt{x=x+2;x=x-1;}]\!] \Rightarrow [\![\texttt{x++;}]\!]$

However, there are also situations where the two relations disagree. This occurs, for example, when a transformation reduces the number of CFG nodes, but increases the execution time. In this case amorphous slicing allows the transformation, but residualisation does not.

The preceding discussion indicates that partial evaluation is both semantically and syntactically different from slicing (though both fit the projection framework defined in Sect. 5). However, there are similarities between the two approaches to program specialisation. Specifically, setting aside the differences between the syntactic ordering for slicing and partial evaluation, we will prove the following statement can be made about the relationship between partial evaluation and conditioned slicing.

Theorem 6.1 Let q be a residual program constructed from a program p, with respect to a set of states Σ If q terminates in all states σ in Σ then q is semantically equivalent to a conditioned slice of p constructed with respect to the slicing criterion $I-(SV_{(p,\Sigma)}, \Sigma)$.

Since partial evaluation is defined in terms of strict semantics, $\mathcal{M}$, and conditioned slicing in terms of lazy semantics $\mathcal{M}_L$, we first need a result which says that a terminating program's lazy and strict meanings are identical:

Proposition 1 Let P be a program and σ be a state in Σ, then $\mathcal{M}[\![P]\!]\sigma \neq \perp \implies \mathcal{M}_L[\![P]\!]\sigma = \mathcal{M}[\![P]\!]\sigma$.

Proof. This is now proved by structural induction over the language being considered.

`skip` **Statement** Trivial as by definition, $\mathcal{M}_L[\![\texttt{skip}]\!]\sigma = \sigma = \mathcal{M}[\![skip]\!]\sigma$ for all σ in Σ.

abort **Statement** The result is vacuously true as, $\mathcal{M}[\![abort]\!]\sigma = \perp$ for all σ in Σ.

Assignment statements. Trivial as $\mathcal{E}_L[\![e]\!]\sigma = \mathcal{E}[\![e]\!]\sigma$ as $\sigma \in \Sigma$.

Conditional statements. Let σ be a state in Σ with $\mathcal{M}[\![\texttt{if}\,(B)\ S_1\ \texttt{else}\ S_2]\!]\sigma \neq \perp$. Then it follows that $\mathcal{E}_L[\![B]\!]\sigma = \mathcal{E}[\![B]\!]\sigma \neq \perp$ as σ is a state in Σ. If $\mathcal{E}_L[\![B]\!]\sigma = \texttt{True}$, then $\mathcal{M}[\![\texttt{if}\,(B)\ S_1\ \texttt{else}\ S_2]\!]\sigma$ is reduced to just $\mathcal{M}[\![S_1]\!]\sigma$ and $\mathcal{M}_L[\![\texttt{if}\,(B)\ S_1\ \texttt{else}\ S_2]\!]\sigma$ is reduced to just $\mathcal{M}_L[\![S_1]\!]\sigma$ and the result follows immediately from the induction hypothesis. Similarly, if $\mathcal{E}_L[\![B]\!]\sigma = \texttt{False}$ as $\mathcal{M}[\![\texttt{if}\,(B)\ S_1\ \texttt{else}\ S_2]\!]\sigma$ is reduced to just $\mathcal{M}[\![S_2]\!]\sigma$ and $\mathcal{M}_L[\![\texttt{if}\,(B)\ S_1\ \texttt{else}\ S_2]\!]\sigma$ is reduced to just $\mathcal{M}_L[\![S_2]\!]\sigma$.

Sequences. Let $\sigma \in \Sigma$ with $\mathcal{M}[\![S_1; S_2]\!]\sigma \neq \perp$. Hence, $\mathcal{M}[\![S_2]\!]\mathcal{M}[\![S_1]\!]\sigma \neq \perp$ and $\mathcal{M}[\![S_1]\!]\sigma \neq \perp$. The result follows immediately by application of the semantics rule for sequences and the induction hypothesis:

$$\begin{aligned}\mathcal{M}_L[\![S_1; S_2]\!]\sigma &= \mathcal{M}_L[\![S_2]\!]\mathcal{M}_L[\![S_1]\!]\sigma (\text{by definition})\\ &= \mathcal{M}[\![S_2]\!]\mathcal{M}[\![S_1]\!]\sigma (\text{induction hypothesis})\\ &= \mathcal{M}[\![S_1; S_2]\!]\sigma.\end{aligned}$$

`While` **loops** Let σ be a state in Σ, such that $[\![\texttt{while}\ (B)\ S]\!]$ terminates in σ. Assume that $[\![\texttt{while}\ (B)\ S]\!]$ terminates in σ after n iterations. Therefore $\mathcal{M}[\![\texttt{while}\ (B)\ S]\!]\sigma = \mathcal{M}[\![\mathcal{W}_i(B, S)]\!]\sigma$ for all $i \geqslant n$.
Thus, using the definition of the lazy meaning of `while` loops, it suffices to show that

$$\forall\, i \geqslant 0,\ \text{if}\ \mathcal{M}[\![\mathcal{W}_i(B, S)]\!]\,\sigma \neq \perp\ \text{then}\ \mathcal{M}_L[\![\mathcal{W}_i(B, S)]\!]\,\sigma = \mathcal{M}[\![\mathcal{W}_i(B, S)]\!]\,\sigma.$$

We show this by induction on i:

[4] This transformation is *allowed* by the definition of partial evaluation, but is not typically exploited in partial evaluation systems.

1. The base case is vacuously true as $\mathcal{M}[\![\mathcal{W}_0(B, S)]\!]\sigma = \bot$.
2. We now assume that the result holds for ith unfolding i.e.:

   $$\forall\, \sigma \in \Sigma,\ \text{if}\ \mathcal{M}[\![\mathcal{W}_i(B, S)]\!]\sigma \neq \bot\ \text{then}\ \mathcal{M}_L[\![\mathcal{W}_i(B, S)]\!]\sigma = \mathcal{M}[\![\mathcal{W}_i(B, S)]\!]\sigma.$$

   Let σ be a state in Σ, with $\mathcal{M}[\![\mathcal{W}_{i+1}(B, S)]\!]\sigma \neq \bot$.

   We must show that $\mathcal{M}_L[\![\mathcal{W}_{i+1}(B, S)]\!]\sigma = \mathcal{M}[\![\mathcal{W}_{i+1}(B, S)]\!]\sigma$.

   If $\mathcal{E}[\![B]\!]\sigma = \texttt{False}$, then $\mathcal{M}_L[\![\mathcal{W}_{i+1}(B, S)]\!]\sigma = \mathcal{M}[\![\mathcal{W}_{i+1}(B, S)]\!]\sigma = \sigma$.

   If, on the other hand, $\mathcal{E}[\![B]\!]\sigma = \texttt{True}$, then

   $$\mathcal{M}[\![\mathcal{W}_{i+1}(B, S)]\!]\sigma = \mathcal{M}[\![\mathcal{W}_i(B, S)]\!]\mathcal{M}[\![S]\!]\sigma$$

   and

   $$\mathcal{M}_L[\![\mathcal{W}_{i+1}(B, S)]\!]\sigma = \mathcal{M}_L[\![\mathcal{W}_i(B, S)]\!]\mathcal{M}_L[\![S]\!]\sigma$$

   and the result follows immediately by the induction hypothesis. □

We are now in a position to prove the main result: Theorem 6.1.

Proof. Let q be a residual program constructed from a program p, with respect to a set of states Σ and assume q terminates in all states σ in Σ.
By Definition 15,

$$\forall\sigma \in \Sigma, \forall v \in (I - SV_{(p,\Sigma)}), \mathcal{M}[\![p]\!]\sigma v = \mathcal{M}[\![q]\!]\sigma v.$$

Since q terminates in all states σ in Σ, Theorem 1 gives,

$$\forall\sigma \in \Sigma, \forall v \in (I - SV_{(p,\Sigma)}), \mathcal{M}_L[\![p]\!]\sigma v = \mathcal{M}_L[\![q]\!]\sigma v.$$

So by Definition 11, $p \approx_{CS}^{(I-SV_{(p,,)}\Sigma)} q$. In other words, q is conditioned equivalent to p with respect to $(I - SV_{(p,,)}\Sigma)$, hence q is semantically equivalent to a conditioned slice of p constructed with respect to the slicing criterion $I - (SV_{(p,\Sigma)}, \Sigma)$, as required. □

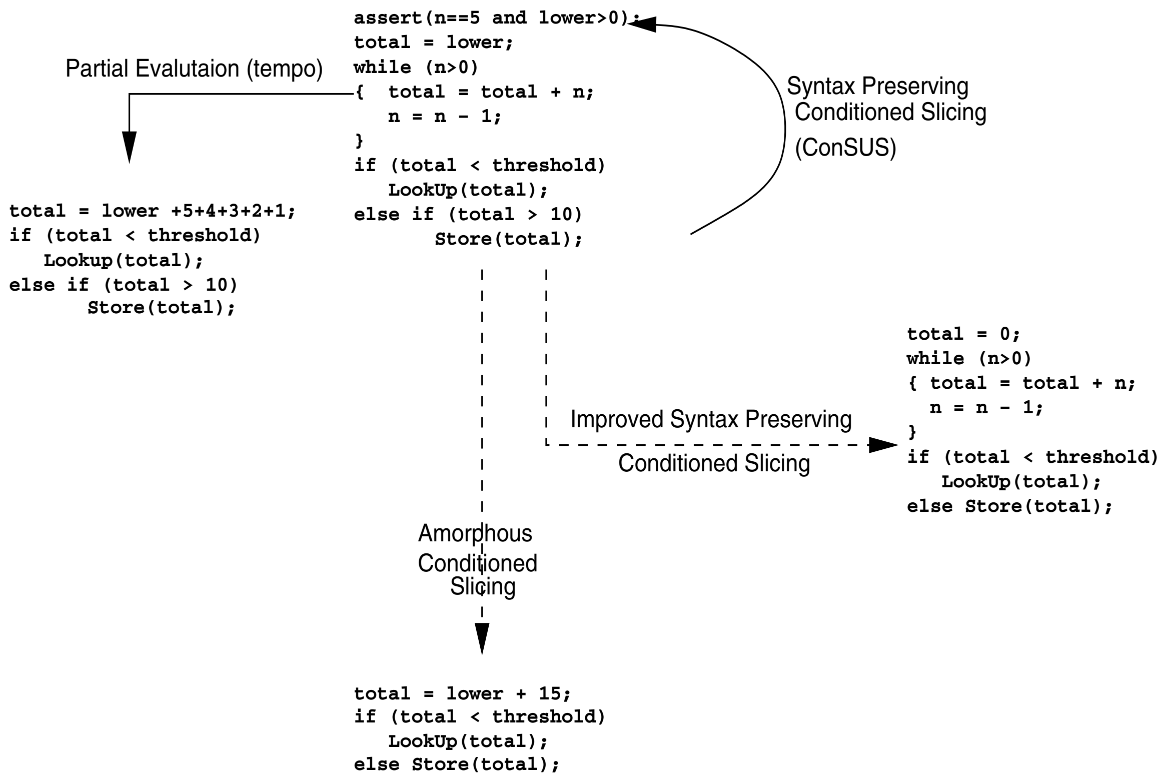
7. Applications

The correspondence between partial evaluation and conditioned slicing established in this paper may have practical applications in program slicing. This section illustrates some of these potential applications, by showing how existing partial evaluation and slicing techniques could be combined in order to improve slicing.

Consider the example in Fig. 8. In this example, there are four program fragments. These will be referred to by points of the compass. The original program is the northerly program fragment. Previous work on conditioned slicing could be applied to this program fragment in an attempt to reduce the program, using knowledge gleaned from the symbolic states of the program as it is symbolically executed. For the purpose of this discussion, it will be supposed that the set of states with respect to which the conditioning takes place is the set of all possible states, which satisfy the assertion at the start of the program.

Applying the existing approaches to conditioning [DDF+04, FDHH04] the program would not be simplified, because current approaches to conditioning handle loops with insufficient precision. For example, using the tool ConSUS [DDF+04], the conditioned slice obtained is the entire program. ConSUS combines symbolic execution with theorem proving to obtain conditioned slices. The symbolic executor works out path conditions, while the theorem prover is used to determine which statements must fail to be executed due to the values of these path conditions. ConSUS can recognise some situations where a loop fails to execute and can exploit limited information about the symbolic execution of the loop, but it fails to detect that the test `if total >10` is redundant. This is because the symbolic execution does not symbolically execute the unfolded body of the loop.

However, if the program were to have been first partially evaluated, then the loop body would have been unfolded. Using this unfolded loop body, the symbolic executor would be able to provide more information


Partial Evalutaion (tempo)
Syntax Preserving Conditioned Slicing (ConSUS)
Improved Syntax Preserving Conditioned Slicing
Amorphous Conditioned Slicing

```
assert(n==5 and lower>0);
total = lower;
while (n>0)
{  total = total + n;
   n = n - 1;
}
if (total < threshold)
   LookUp(total);
else if (total > 10)
      Store(total);
```

```
total = lower +5+4+3+2+1;
if (total < threshold)
   Lookup(total);
else if (total > 10)
      Store(total);
```

```
total = 0;
while (n>0)
{ total = total + n;
  n = n - 1;
}
if (total < threshold)
   LookUp(total);
else Store(total);
```

```
total = lower + 15;
if (total < threshold)
   LookUp(total);
else Store(total);
```

Fig. 8. Combining partial evaluation and slicing techniques

about the terminating state of the loop to the theorem prover, allowing it to determine that the test `if total > 10` is redundant. This would lead to the improved syntax-preserving conditioned slice in the easterly quadrant. This conditioned slice is thinner than that produced by any existing tool for conditioned slicing, showing that the combination of partial evaluation and conditioned slicing can produce improvements in existing conditioned slicing approaches.

A more dramatic improvement in slicing technology could be obtained by exploiting the amorphous nature of partial evaluation. Amorphous slicing is known to reduce the size of slices because it allows transformation to be applied [De 01, DDF$^+$04, HBD03, War02]. This is to be contrasted with syntax-preserving slicing which allows only statement deletion.

It would be desirable to construct a tool for amorphous conditioned slicing. Such a tool would improve the simplification power of static syntax-preserving slicing, both because it would be amorphous (not syntax-preserving) and because it would be conditioned (not static). Hitherto, no such slicing tool has been constructed; it has proved hard to find effective algorithms for syntax-preserving conditioned slicing, so amorphous conditioned slicing would seem to be an even harder problem. Ward [War03] has shown how a form of amorphous conditioned slicing (called a semantic slice) could be produced by combining the transformations of FermaT [War99] with slicing, but he gives no general algorithm for semantic slicing. The discovery of effective algorithms for amorphous conditioned slicing therefore remains an open problem.

However, some combination of amorphous static slicing, syntax-preserving conditioned slicing and partial evaluation may yield techniques for amorphous conditioned slicing. Consider the program in the southerly quadrant of Fig. 8. This program is an amorphous conditioned slice of the original program in the northerly quadrant. Currently, no slicing tool is available which can produce this program and there are no algorithms in the literature for amorphous conditioned slicing.

Applying partial evaluation (using `tempo` [CHM$^+$98b]), to the original program in the northerly quadrant, the residual program in the westerly quadrant is obtained. This program is an amorphous conditioned slice (as the previous sections have demonstrated). Unsurprisingly, it is not the ideal formulation of the syntax of the amorphous slice in the southerly quadrant, because `tempo` is designed for partial evaluation, not slicing. However,

Equivalent to *wc -lc*	Equivalent to *wc -c*	Equivalent to *wc -l*
`void line_char_count`	`void line_char_count`	`void line_char_count`
`    (FILE *f)`	`    (FILE *f)`	`    (FILE *f)`
`{`	`{`	`{`
`int lines = 0;`	▭	`int lines = 0;`
`int chars;`	`int chars;`	▭
`BOOLEAN eof_flag = FALSE`	`BOOLEAN eof_flag = FALSE`	`BOOLEAN eof_flag = FALSE`
`int n;`	`int n;`	▭
`extern void`	`extern void`	`extern void`
`scan_line(`	`scan_line(`	`scan_line(`
`          FILE *f,`	`          FILE *f,`	`          FILE *f,`
`          BOOLEAN *bptr,`	`          BOOLEAN *bptr,`	`          BOOLEAN *bptr,`
`          int *iptr);`	`          int *iptr);`	▭`);`
`scan_line(f,&eof_flag,&n);`	`scan_line(f,&eof_flag,&n);`	`scan_line(f,&eof_flag,&n);`
`chars = n;`	`chars = n;`	▭
`while(eof_flag == FALSE) {`	`while(eof_flag == FALSE) {`	`while(eof_flag == FALSE) {`
`  lines = lines + 1;`	▭	`  lines = lines + 1;`
`  scan_line(f,&eof_flag,&n);`	`  scan_line(f,&eof_flag,&n);`	`  scan_line2(f,&eof_flag,`▭`);`
`  chars = chars + n;`	`  chars = chars + n;`	▭
`}`	`}`	`}`
`printf("lines=",lines);`	▭	`printf("lines=",lines);`
`printf("chars=",chars);`	`printf("chars=",chars);`	▭
`}`	`}`	`}`

Fig. 9. A scaled-down word-count and two slice-like specialisation

it is tantalisingly close. Using amorphous slicing the expression 5+4+3+2+1 would be (trivially) transformed to 15. Using traditional syntax-preserving conditioned slicing, the constraint `lower >0` could be exploited, in combination with symbolic state information to determine that the test `if total >10` is redundant. This last step is nothing more than generalised partial evaluation, which was not supported by `tempo` at the time of writing.

The primary role of this paper is theoretical; it demonstrates the correspondence between slicing and partial evaluation. However, the importance of this result is likely to be the application of the connection between the two in work on practical techniques for computing slices. The observations in this section show that it may be possible to combine existing techniques for amorphous slicing and partial evaluation to produce amorphous conditioned slices; a form of slicing for which there are no currently existing techniques.

8. Related work

The relationship between partial evaluation and slicing was first (informally) considered by Reps and Turnidge [RT96]. They concluded

"*The two approaches are actually* complementary*: Partial evaluation and its forwards-orientated relatives take information known at the* beginning *of a program and push it* forward*; backward [static] slicing takes a demand for information at the* end *of a program and pushes it backward.*"[RT96]

They provide as an example the program and two of its slices shown in Fig. 9. The example concerns a scaled-down version of the UNIX word-count utility, which computes the number of lines and number of characters in a file, `f`, passed as a parameter. The file `f` is considered to be the input to the program. The program is written so that the input contains no information as to whether or not lines or characters are to be counted, so the program will compute both regardless of the input.

Partial evaluation cannot therefore be applied to this program to separate out the two computations with respect to the variables `lines` and `chars`. Syntax-preserving end-slicing can do this however. The syntax-preserving end-slices on the values of `chars` and `lines` are depicted in the central and rightmost sections of Fig. 9 respectively.

Reps and Turnidge [RT96] were interested in syntax-preserving slicing, rather than amorphous or conditioned slicing. From Definitions 8 and 11 it can be seen that semantically, an amorphous slice is simply a conditioned slice where the the set of states Σ, is the set of all possible initial states. From Theorem 6.1 it thus follows that if q is a residual program constructed from a program p, with respect to a set of all states Σ and if q terminates in all states σ in Σ then q is semantically equivalent to an amorphous slice of p constructed with respect to the non-static variables of p.

For the example suggested by Reps and Turnidge, it is not possible to remove computation on either `chars` or `lines` individually, because both depend upon the same dynamic variable. Making this variable static will thus make both computations disappear. There is no way to use partial evaluation, as currently implemented, to produce an amorphous slice on the individual variables of the program.

9. Summary

This paper highlights the different syntactic and semantic interpretations which underly slicing and partial evaluation. Various forms of slicing and partial evaluation are formulated in terms of a program projection framework. Having expressed both slicing and partial evaluation using identical terminology, it is then possible formally to compare the two techniques. It is proved that for terminating programs, a residual program produced by partial evaluation is semantically equivalent to a conditioned slice.

References

[And92] Andersen L (1992) Self-applicable C program specialization. In: Proceedings of the 1992 ACM workshop on partial evaluation and semantics-based program manipulation, San Francisco, USA., June 1992. Association for Computing Machinery, pp 54–61

[BE93] Beck J, Eichmann D (1993) Program and interface slicing for reverse engineering. In: Proceedings of the IEEE/ACM 15th conference on software engineering (ICSE'93), IEEE Computer Society Press, Los Alamitos, California, USA, pp 509–518

[BF98] Blazy S, Facon P (1998) Partial evaluation for program comprehension. ACM Comput Surveys 30(3es), Article 17

[BG96] Binkley DW, Gallagher KB (1996) Program slicing. In: Zelkowitz M (ed) Advances in computing, vol. 43, Academic, New York, pp 1–50

[BH04] Binkley DW, Harman M (2004) A survey of empirical results on program slicing. Adv Comput 62:105–178

[BHO$^+$76] Beckman L, Haraldson A, Oskarsson O, Sandewall E (1976) A partial evaluator, and its use as a programming tool. Artif Intell 7(4):319–357

[Bin97] Binkley DW (1997) Semantics guided regression test cost reduction. IEEE Trans Softw Eng 23(8):498–516

[Bin98] Binkley DW (1998) The application of program slicing to regression testing. In: Harman M, Gallagher K (eds), Information and software technology special issue on program slicing, vol. 40, Elsevier, USA, pp 583–594

[Bin99] Binkley DW (1999) Computing amorphous program slices using dependence graphs and a data-flow model. In: ACM symposium on applied computing, The Menger, San Antonio, Texas, USA, ACM Press, New York, NY, USA, pp 519–525

[BO94] Bieman JM, Ott LM (1994) Measuring functional cohesion. IEEE Trans Softw Eng 20(8):644–657

[CCD98] Canfora G, Cimitile A, De Lucia A (1998) Conditioned program slicing. In: Harman M, Gallagher K (eds) Information and software technology special issue on program slicing, vol 40, Elsevier Science B. V., USA, pp 595–607

[CDLM96] Cimitile A, De Lucia A, Munro M (1996) A specification driven slicing process for identifying reusable functions. Softw Maint: Res Pract 8:145–178

[CF89] Cartwright R, Felleisen M (1989) The semantics of program dependence. In: Proceedings of ACM SIGPLAN conference on programming language design and implementation pp 13–27

[CHM$^+$98a] Consel C, Hornof L, Marlet R, Muller G, Thibault S, Volanschi E-N (1998) Partial evaluation for software engineering. ACM Computing Surveys, 30(3es), September 1998. Article 20

[CHM$^+$98b] Consel C, Hornof L, Marlet R, Muller G, Thibault S, Volanschi E-N (1998) Tempo: specializing systems applications and beyond. ACM Computing Surveys, 30(3es), Article 19

[Das98] Das M (1998) Partial evaluation using dependence graphs. PhD thesis, University of Wisconsin–Madison

[DDF$^+$04] Danicic S, Daoudi M, Fox C, Harman M, Hierons RM, Howroyd J, Ouarbya L, Ward M (2004) Consus: A lightweight program conditioner. J Syst Softw (accepted)

[DDH$^+$02] Daoudi M, Danicic S, Howroyd J, Harman M, Fox C, Ouarbya L, Ward M (2002) ConSUS: A scalable approach to conditioned slicing. In: Proceedings of IEEE working conference on reverse engineering (WCRE 2002), Richmond, Virginia, USA, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA, pp 109 – 118 (Invited for special issue of the Journal of Systems and Software as best paper from WCRE 2002)

[DDH04] Danicic S, De Lucia A, Harman M (2004) Building executable union slices using conditioned slicing. In: Proceedings of the 12th international workshop on program comprehension (IWPC 2004), Bari, Italy, June 2004. IEEE Computer Society Press, Los Alamitos, California, USA, pp 89–97

[De 01] De Lucia A (2001) Program slicing: Methods and applications. In: Procedings of the 1st IEEE international workshop on source code analysis and manipulation, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA, pp 142–149

[DFHH00] Danicic S, Fox C, Harman M, Mark Hierons R (2000) ConSIT: A conditioned program slicer. In: Proceedings of the IEEE international conference on software maintenance (ICSM'00), San Jose, California, USA, October 2000. IEEE Computer Society Press, Los Alamitos, California, USA, pp 216–226

[DFM96] De Lucia A, Fasolino AR, Munro M (1996) Understanding function behaviours through program slicing. In: Proceedings of the 4th IEEE workshop on program comprehension, Berlin, Germany, March 1996. IEEE Computer Society Press, Los Alamitos, California, USA, pp 9–18

[DHHO04] Danicic S, Harman M, Howroyd J, Ouarbya L (2004) A lazy semantics for program slicing. In: Proceedings of the $1^{st.}$ international workshop on programming language interference and dependence, Verona, Italy

[DHN98] Dwyer M, Hatcliff J, Nanda M (1998) Using partial evaluation to enable verification of concurrent software. ACM Computing Surveys, 30(3es), September 1998. Article 22

[Dra98] Draves S (1998) Partial evaluation for media processing. ACM Computing Surveys, 30(3es), September 1998. Article 21

[Ers78] Ershov AP (1978) On the essence of computation. North-Holland, Amsterdam, pp 391–420

[FDHH04] Fox C, Danicic S, Harman M, Mark Hierons R (2004) ConSIT: a fully automated conditioned program slicer. Softw Pract Exp 34:15–46, (Published online 26th November 2003)

[FHHD01] Fox C, Harman M, Mark Hierons R, Danicic S (2001) Backward conditioning: a new program specialisation technique and its application to program comprehension. In: Proceedings of the 9th IEEE international workshop on program comprenhesion (IWPC'01), Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, California, USA, pp 89–97

[FN87] Futamura Y, Nogi K (1987) Generalized partial computation. In: Bjørner D, Ershov AP, Jones ND (eds) IFIP TC2 Workshop on partial evaluation and mixed computation, North–Holland, pp 133–151

[FRT95] Field J, Ramalingam G, Tip F (1995) Parametric program slicing. In: Proceedings of the 22nd ACM symposium on principles of programming languages, San Francisco, CA, pp 379–392

[Fut71] Futamura Y (1971) Partial evaluation of computation process – an approach to a compiler compiler. Syst Comput Controls 2(5):721–728

[Fut99a] Futamura Y (1999) Partial evaluation of computation process – an approach to a compiler-compiler. HighOrder Symbolic Comput 12(4):381–391 (Reprinted from Syst · Comput · Controls 2(5), 1971, with a foreword)

[Fut99b] Futamura Y (1999) Partial evaluation of computation process, revisited. HighOrder Symbolic Comput 12(4):377–380

[GHS92] Gupta R, Jean Harrold M, Lou Soffa M (1992) An approach to regression testing using slicing. In: Proceedings of the IEEE conference on software maintenance, Orlando, Florida, USA, 1992. IEEE Computer Society Press, Los Alamitos, California, USA, pp 299–308

[GJ89] Gomard CK, Jones ND (1989) Compiler generation by partial evaluation. In: Ritter GX (ed) Information processing '89. proceedings of the IFIP 11th World computer congress, IFIP, Amsterdam: North-Holland, pp 1139–1144

[GJ91] Gomard CK, Jones ND (1991) Compiler generation by partial evaluation: a case study. Struct Program 12:123–144

[GL91] Gallagher KB, Lyle JR (1991) Using program slicing in software maintenance. IEEE Trans Softw Eng 17(8):751–761

[GM03] Giacobazzi R, Mastroeni I (2003) Non–standard semantics for program slicing. HighOrder Symbolic Comput 16(4):297–339 (Special issue on partial evalution and semantics-based program manipulation)

[GS96] Glück R, Sørensen M (1996) A roadmap to metacomputation by supercompilation. In: Danvy O, Glück R, Thiemann P (eds) Proceedings of the Dagstuhl seminar on partial evaluation, vol 1110, Schloss Dagstuhl, Wadern, Germany, 12–16 February 1996. Springer, Berlin Heidelberg New York, pp 137–160

[Har78] Haraldsson A (1978) A partial evaluator, its use for compiling iterative statements in Lisp. In: Conference record of the 5th annual ACM symposium on principles of Programming Languages, Tucson, Arizona, pp 195–202

[HBD03] Harman M, Binkley DW, Danicic S (2003) Amorphous program slicing. J Syst Softw 68(1):45–64

[HD95] Harman M, Danicic S (1995) Using program slicing to simplify testing. Softw Test Verif Rel 5(3):143–162

[HD97] Harman M, Danicic S (1997) Amorphous program slicing. In: Proceedings of the $5th$ IEEE international workshop on program comprenhesion (IWPC'97), Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA, pp 70–79

[HH01] Harman M, Hierons RM (2001) An overview of program slicing. Softw Focus 2(3):85–92

[HHD99] Hierons RM, Harman M, Danicic S (1999) Using program slicing to assist in the detection of equivalent mutants. Softw Test Verif Rel 9(4):233–262

[HHD$^+$01] Harman M, Hierons RM, Danicic S, Howroyd J, Fox C (2001) Pre/post conditioned slicing. In: Proceedings of the IEEE international conference on software maintenance (ICSM'01), Florence, Italy, November 2001. IEEE Computer Society Press, Los Alamitos, California, USA, pp 138–147

[HHF$^+$02] Hierons RM, Harman M, Fox C, Ouarbya L, Daoudi M (2002) Conditioned slicing supports partition testing. Softw Test Verif Rel 12:23–28

[HHZM01] Harman M, Hu L, Zhang X, Munro M (2001) GUSTT: an amorphous slicing system which combines slicing and transformation. In: Proceedings of the $1st$ Workshop on Analysis, Slicing, and Transformation (AST 2001), Stuttgart, October 2001. IEEE Computer Society Press, Los Alamitos, California, USA, pp 271–280

[HPR89] Horwitz S, Prins J, Reps T (1989) Integrating non–interfering versions of programs. ACM Trans Program Lang Syst 11(3):345–387

[HRB88] Horwitz S, Reps T, Binkley DW (1988) Interprocedural slicing using dependence graphs. In: ACM SIGPLAN conference on programming language design and implementation, Atlanta, Georgia, Proc SIGPLAN Notices 23(7):35–46

[JGS93] Jones ND, Gomard CK, Sestoft P (1993) Partial evaluation and automatic program generation. Prentice-Hall, London, England

[JR94] Jackson D, Rollins EJ (1994) Chopping: A generalisation of slicing. Technical Report CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

[JSS85] Jones ND, Sestoft P, Søndergaard H (1985) An experiment in partial evaluation: The generation of a compiler generator. In: Jouannaud J-P (ed) Rewriting techniques and applications: Dijon, France, vol 202 of Lecture notes in computer science, Springer Berlin Heidelberg New York, pp 124–140

[JSS89] Jones ND, Sestoft P, Søndergaard H (1989) Mix: A self-applicable partial evaluator for experiments in compiler generation. Lisp Symbol Comput 2(1):9–50 (DIKU Report 91/12)

[Kam93] Kamkar M (1993) Interprocedural dynamic slicing with applications to debugging and testing. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, Available as Linköping Studies in Science and Technology, Dissertations, Number 297

[KKZG95] Kleinrubatscher P, Kriegshaber A, Zöchling R, Glück R (1995) Fortran program specialization. SIGPLAN Notices 30(4):61–70

[KS98] Krinke J, Snelting G (1998) Validation of measurement software as an application of slicing and constraint solving. In: Harman M, Gallagher K (eds) Information and software technology special issue on program slicing, vol 40, Elsevier, USA, pp 661–675

[Lak93] Lakhotia A (1993) Rule–based approach to computing module cohesion. In: Proceedings of the 15*th* conference on software engineering (ICSE-15), pp 34–44

[LCYK01] Lee WK, Chung IS, Yoon GS, Kwon YR (2001) Specification-based program slicing and its applications. J Syst Architect 47:427–443

[LD98] Lakhotia A, Deprez J-C (1998) Restructuring programs by tucking statements into functions. In: Harman M, Gallagher K (eds) Information and software technology special issue on program slicing, vol 40, Elsevier, USA, pp 677–689

[LE93] Liu L, Ellis R (1993) An approach to eliminating COMMON blocks and deriving ADTs from Fortran programs. Technical report, University of Westminster, UK

[LW87] Lyle JR, Weiser M (1987) Automatic program bug location by program slicing. In: Proceedings of the 2nd international conference on computers and applications, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California, USA, pp 877–882

[OO84] Ottenstein KJ, Ottenstein LM (1984) The program dependence graph in software development environments. SIGPLAN Notices 19(5):177–184

[OT93] Ott LM, Thuss JJ Slice based metrics for estimating cohesion. In: Proceedings of the IEEE-CS international metrics symposium, Baltimore, Maryland, USA, May 1993. IEEE Computer Society Press, Los Alamitos, California, USA, pp 71–81

[Oua05] Ouarbya L (2005) A lazy semantics for program slicing. PhD Thesis, Department of Computing, Goldsmiths College, University of London (To appear)

[RT96] Reps T, Turnidge T (1996) Program specialization via program slicing. In: Danvy O, Glück R, Thiemann P (eds) In: Proceedings of the Dagstuhl seminar on partial evaluation, vol 1110 Schloss Dagstuhl, Wadern, Germany, Springer, Berlin Heidelberg New York, NY, pp 409–429

[SGJ94] Sørensen MH, Glück R, Jones ND (1994) Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In: Sannella D (ed) Programming languages and systems—ESOP'94, 5th European symposium on programming, vol 788 of Lecture notes in computer science, Edinburgh, U.K, Springer, Berlin Heidelberg New York, pp 485–500

[Sha91] Shahmehri N (1991) Generalized algorithmic debugging. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden. Available as Linköping Studies in Science and Technology, Dissertations, Number 260

[Ste75] Stefanescu DC (1975) The definition of a LISP compiler using partial evaluation. Report, M.I.T., Department of EE & CS, Cambridge, Massachusetts, July 1975. BS Thesis

[SVM$^+$93] Simpson D, Valentine SH, Mitchell R, Liu L, Ellis R (1993) Recoup – Maintaining Fortran. ACM Fortran forum 12(3):26–32

[Tip95] Tip F (1995) A survey of program slicing techniques. J Program Lang 3(3):121–189

[War99] Ward M (1999) Assembler to C migration using the FermaT transformation system. In: Proceediongs of IEEE international conference on software maintenance (ICSM'99), Oxford, UK, IEEE Computer Society Press, Los Alamitos, California, USA

[War02] Ward M (2002) Program slicing via FermaT transformations. In: Proceedings of the 26th IEEE annual computer software and applications conference (COMPSAC 2002), Oxford, UK, August 2002. IEEE Computer Society Press, Los Alamitos, California, USA, pp 357–362

[War03] Ward M (2003) Slicing the SCAM mug: A case study in semantic slicing. In: Proceedings of the IEEE international workshop on source code analysis and manipulation (SCAM 2003), Amsterdam, Netherlands, September 2003. IEEE Computer Society Press, Los Alamitos, California, USA, pp 88–97

[Wei79] Weiser M (1979) Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, University of Michigan, Ann Arbor, MI, USA

[Wei82] Weiser M (1982) Programmers use slicing when debugging. Commun ACM 25(7):446–452

Received May 2002
Revised Octber 2004 and July 2005
Accepted August 2005 by D. J. Cooke
Published online May 10 2006