A Trajectory-based Strict Semantics for Program Slicing

Richard W. Barraclough [b] David Binkley [a] Sebastian Danicic [b]
Mark Harman [c] Robert M. Hierons [d] Ákos Kiss [e]
Mike Laurence [b] Lahcen Ouarbya [b]

[a] *Computer Science Dept., Loyola College, Baltimore MD 21210-2699, USA.*

[b] *Dept. of Computing, Goldsmiths College, University of London, London SE14 6NW, UK.*

[c] *Dept. of Computer Science, King's College London WC2R 2LS, UK.*

[d] *School of Information Systems, Computing, and Mathematics, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK.*

[e] *Dept. of Software Engineering, Institute of Informatics, University of Szeged, 6720 Szeged, Hungary.*

Abstract

In this paper we define a simple, intuitive trajectory-based equivalence relation which accurately captures the relationship between programs and the slices produced by the most commonly used and well-known slicing algorithms, Weiser's and the PDG algorithm, even for non-terminating programs.

Our new equivalence, unlike other approaches, is based on standard semantics. It deals with non-termination by stipulating that a program and its slice must 'agree' in all terminating 'approximations' based on finite loop unfoldings. We prove that this equivalence is preserved by Weiser's algorithm and all slicing algorithms based on data and control dependence. We also demonstrate that this new equivalence more accurately captures the behaviour of slicing algorithms than previous work.

1 Introduction

Program slicing is a technique for extracting an executable sub-program – the slice – from a larger program. In order to be useful the slice must preserve some interesting properties of the program, which is usually the final values of some set of variables. This is a semantic relationship between the program and the slice.

Non-terminating programs present problems to slicing theories and algorithms. These problems arise in trying to define the semantic relationship that slicing must preserve in the case of non-termination. The fact that termination is undecidable does not help matters.

Many slicing theories simply ignore non-terminating programs, but this is increasingly unacceptable since programs where non-termination is both normal and desirable – such as reactive systems – are increasingly common. As we shall see, existing slicing theory faces many problems when non-termination is introduced, but the associated slicing algorithms often produce sensible slices. This suggests that slicing non-terminating programs should be a natural extension to existing work.

In this paper we first, in section 2, examine current work on program slicing. In Section 3 we examine some other approaches to defining semantic equivalence between non-terminating programs, and show that they do not have the important substitutivity property. Some of the problems we highlight here further motivate our new definition of semantic equivalence.

In section 4 we recap work on trajectory semantics and in section 5 we examine in detail how it could be 'patched up' to work for non-terminatong programs.

At last in Section 6 we define our new trajectory semantics, and in Section 7 we prove that it induces the required semantic equivalence relation.

In Section 8 we prove the main result – that data dependence and control dependence are respected by our equivalence relation. Thus dependence based slicing algorithms preserve our relation. In Section 9 we assess our new equivalence by comparing it with the related work in Section 3, and finish in Section 10 with our conclusions and some areas for future work.

2 Background

Comprehensive surveys of program slicing, applications, techniques and results can be found in several survey papers [4,5,15,21,34,39]. There are many forms of program slicing: Static [37], dynamic [27] and conditioned [7]; forward and backward [25]; amorphous and syntax preserving [19]; non-termination removing [35,36] and non-termination preserving [31].

This paper focusses on static backward slicing and extends the authors' previous work [1,2,10,11]. Static backward slicing is the most widely studied slicing paradigm and the one for which previous attempts at formalisation have paid greatest attention [8]. Definition 4.6 gives a formal definition.

2.1 Slicing Algorithms

Historically, slicing was developed first as an algorithm, and only subsequently as a formal definition; practice has preceded theory. Applications of slicing include program comprehension [19], software maintenance [17], testing and debugging [3,22], virus detection [29], integration [6], refactoring [26], restructuring, reverse engineering and reuse [7].

The most well-known and widely used slicing algorithms are variants of Weiser's Algorithm [37] and the PDG algorithm of Horwitz and Reps [24]. These algorithms are essentially the same and produce identical results and they are both based on the notions of data and control dependence [16].

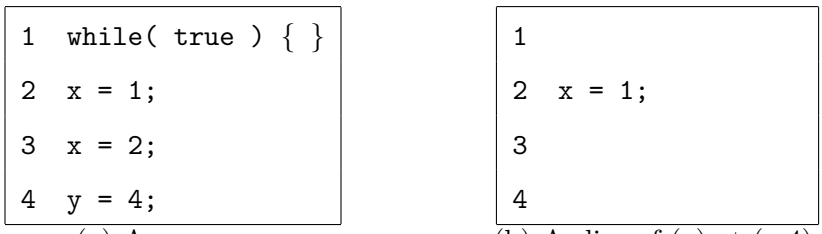
2.2 Non-termination

Recent developments in the application of slicing to reactive programs [23] and finite state machine models [28] requires slices of programs and models that may fail to terminate. This has led to new definitions of control flow for such non-terminating programs and algorithms for computing slices using them [30,31]. This interest in slicing non-terminating programs provides renewed impetus to the search for a sound theoretical foundation for slicing non-terminating programs. The problem of correctly and precisely accounting for the behaviour of slices of non-terminating programs remains a current topic of research.

2.3 Comparing Slicing Algorithms

To be useful a slicing algorithm must preserve some semantic relationship between the original program and its slice produced by the algorithm. Using this semantic relationship the authors introduced [1,2,10,11] a mathematical framework which allows properties of different slicing algorithms to be formally defined and compared.

A shortcoming of the previously-defined framework is that the semantic relationships it uses are not defined for programs that do not terminate for some inputs. The theory does not say anything about the relationship that is preserved by slicing algorithms in these cases: We do not know whether it is preserved or not. Of course, we do not generally know *a priori* whether or not the program being sliced falls into this 'awkward' category: This is a very well-known undecidable problem.



(a) A program. (b) A slice of (a) at (x,4).

Fig. 1. Using Ward and Zedan's refinement-based slicing, (b) is a slice of (a) with respect to the criterion (x,4). Their definition allows any non-terminating program to have any sub-program as a slice because any program is a refinement of a non-terminating program.

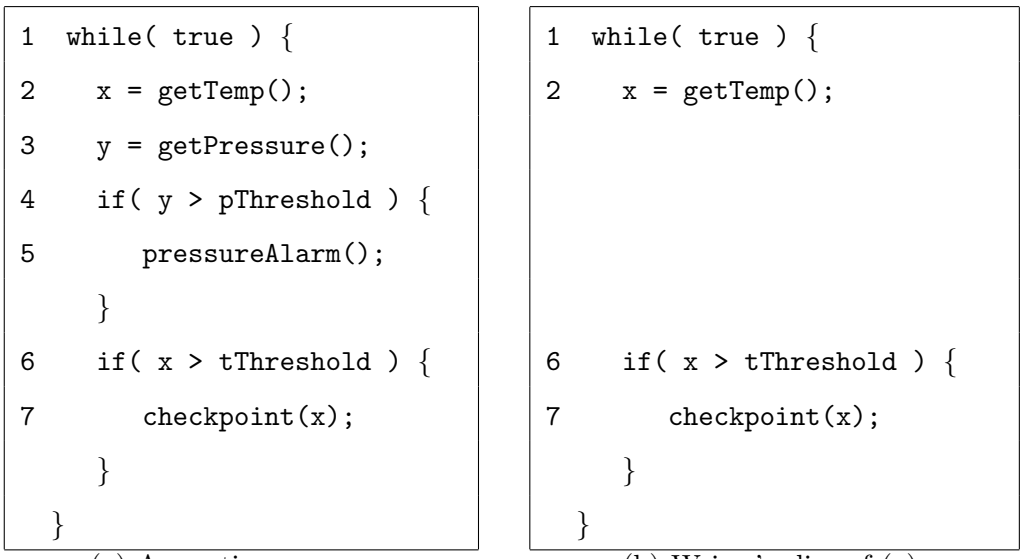
3 Related Work And Applicability To Non-Terminating Programs

Defining precisely the semantics preserved by slicing has proved problematic, largely owing to the possibility of infite loops – intentional or otherwise. Weiser recognised these difficulties in his seminal 1984 paper on slicing [38], and circumvented the problem by defining a slice for terminating executions only. Reps and Yang later showed [**?**] that for any initial state on which the program halts, the program and Weiser's slice compute the same sequence of values for each variable in the slice. Thus for terminating programs the situation is well understood.

3.1 The Refinement Approach

Ward and Zedan [36] argue that slicing should be defined semantically as a 'refinement.' They define a refinement of a program p to be any program that dominates p, thereby allowing slicing to replace any non-terminating computations with arbitrary terminating computations. They also argued that slices should be syntactically amorphous (following Harman and Danicic [19]), but it is the semantic requirement that is important here and their semantic observations regarding refinement apply equally well to syntax-preserving slices as they do to amorphous slices.

Ward and Zedan's refinement-based definition allows some rather odd slices, for example as shown in Figure 1. The problem is that any program is a refinement of a non-terminating program. The approach therefore only imprecisely captures the behaviour of existing slicing algorithms in the presence of non-termination: It imposes the weakest possible requirement. Even worse, the fact that any program is a slice means that slicing by this definition gives no information about non-terminating programs.



(a) A reactive program. (b) Weiser's slice of (a).

Fig. 2. A reactive program (a) and Weiser's slice of it (b). Although (a) is very simple, it is typical of the kinds of infinite computation found in reactive systems. Weiser's slice (b) at line 7 is sensible, even though Weiser's slicing algorithm only applies to terminating programs. Thus we would like the slicing of non-terminating programs to be a logical extension of slicing terminating programs.

3.2 Extending Algorithms For Terminating Programs

Despite the fact that Weiser made no claims for the behaviour of his slicing algorithm in the presence of non-termination, several authors noticed that slices produced from a non-terminating program by Weiser's algorithm (and the other slicing algorithms that followed) do behave in a consistent, compelling and meaningful manner – see Figure 2 for an example. Here the behaviour of slicing algorithms for non-terminating programs is merely a logical extension of their behaviour for terminating programs, allowing meaningful slices like the one in Figure 2(b).

This observation has led several authors to attempt to capture the semantics preserved by program slicing for non-terminating programs: The task is to define a semantics and show equivalence of the program and its slice under the semantics.

3.3 Lazy Semantics

Cartwright and Felleisen [8] were the first to try to generalise. They argued that it is possible to adopt a lazy view of the semantics of imperative programs. Under this lazy interpretation the program in Figure 3(c) fails to terminate

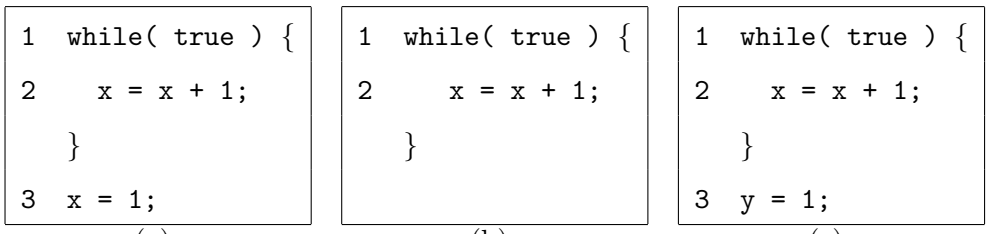


Fig. 3. Both the lazy semantics of Cartwright and Felleisen [8] and the transfinite computation semantics of Giacobazzi and Mastroeni [18] allow the value of `x` to be computed in program (a), but leave it undefined in program (b). No value is computed for `x` in program (c) but the final value of `y` is computed as 1.

on the computation of the variable `x`, but it terminates on the computation of `y`. Under the strict semantics, of course, the program simply fails to terminate and the final values of both `y` and `x` are undefined. However, Cartwright and Felleisen argued that the state update operation could be re-formulated to allow undefined values to be overwritten by defined values. In so doing, they created a lazy interpretation of the computation of imperative languages.

This creates a necessary semantic condition for a program q to be a slice of a program p – equivalence under the lazy semantics. Using this condition to define a slice allows its application to non-terminating programs, so that slicing can be meaningfully applied to reactive systems and other purposefully non-terminating programs.

While it is traditional to view the semantics of imperative programs as strict [33], there is no reason not to consider what it would mean to interpret imperative semantics in a lazy manner. Cartwright and Felleisen argued that in static analysis of program dependences, this is precisely the interpretation that is implicitly adopted.

3.4 Transfinite Computation

Giacobazzi and Mastroeni also defined a non-standard semantics for programs [18] in order to attempt to capture the semantic behaviour of program slicing for non-terminating programs. Their approach does not use a lazy approach but instead defines a semantics in which transfinite computation is possible. In this interpretation, non-termination is not possible, though computations may be infinite. That is, conceptually, computation may continue after a non-terminating loop. See Figure 3 for an example of how this can help in some cases. However, as we shall see in the next section, this complicates matters when the control of the program after an infinite loop is dependent on values computed (or, rather, left undefined) by the infinite loop.

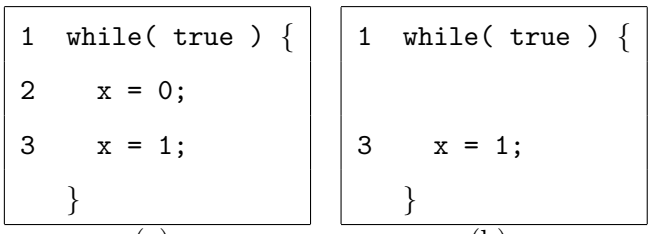


Fig. 4. When using the semantics of Giacobazzi and Masteroeni [18], the final value of the variable x when executing the program (a) is undefined but in (b) it is 1. The semantics of Nestra [**?**], however, the final value of x is 1 in both cases.

Nestra [**?**] defined a new transfinite semantics based on the semantics of Giacobazzi and Mastroeni [18]. Unlike the semantics of Giacobazzi and Mastroeni [18] where all the states observed during the infinite execution are used, Nestra [**?**] has shown that it suffices to consider a subset of these states. Nestra [**?**] shows that there is no need to consider the traces resulting from the execution of each atomic statement inside an infinite loop and that it is enough to consider only the states observed at the top point of the loop. Nestra [**?**] provided a theoretical background to show that the use of these states is enough to capture the standard semantic anomaly. Figure 4 illustrates the difference.

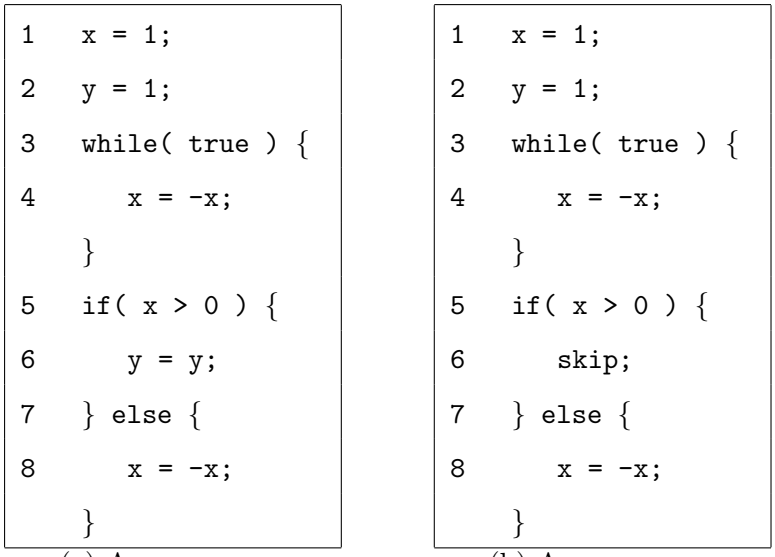
3.5 Substitutive Semantics

Substitutivity means that we can substitute a part of a program by a semantically equivalent part and still preserve the semantics of the original program.

Definition 3.1 (Substitutivity) *A semantics is substitutive if and only if whenever a sub-program q of a program p is replaced with another semantically equivalent program, q', the resulting program p' is semantically equivalent to p.*

Substitutivity is a very natural, and some might argue, essential property of a semantics as it makes correctness proofs of transformation algorithms much more straightforward [19,20], especially if the algorithm is expressed denotationally. Danicic et al. [9] have shown that neither the semantics of Giacobazzi and Mastroeni [18] nor that of of Cartwright and Felleisen [8] is substitutive. In Theorem 7.2 we show that our finite trajectory semantics is substitutive.

The situation is illustrated in Figure 5 by the substitution on line 6. In program p, the value of the variable `x` after executing the infinite loop is undefined, and thus, so is the value of the `if` predicate. Therefore the final value of the variable `y` demands the evaluation of an undefined predicate. For this reason the final value of `y` is undefined when using either the semantics of Giacobazzi and



(a) A program, p. (b) A program, q.

Fig. 5. The programs (a) and (b) are not equivalent under either Cartwright and Felleisen's lazy semantics or under Giacobazzi and Mastroeni's transfinite semantics. For program (a) both semantics give the final value of `y` as 'undefined.' For program (b) they both give the final value of `y` to be 1. However, the statements `y=y` and `skip` are semantically equivalent. Therefore neither of these semantics is substitutive.

Mastroeni [18], that of Nestra [**?**], or that of Cartwright and Felleisen [8].

The program q in Figure 5(b) is obtained by replacing the statement `y=y;` on line 6 of p with the semantically equivalent `skip`. However, when executing q, the semantics of Giacobazzi and Mastroeni [18], the semantics of Nestra [**?**], and the semantics of Cartwright and Felleisen [8] give the final value of `y` to be 1. This shows that none of these three semantics are substitutive.

The non-substitutivity of the semantics of Cartwright and Felleisen and Giacobazzi and Mastroeni was addressed by Danicic et al. [9] with a modified lazy semantics for slicing at the end of a program. Under the lazy semantics of Danicic et al., the final value of `y` is the same as its initial state, i.e., 1, for both versions of the program. While this approach correctly accounts for the behaviour of Weiser's slicing algorithm for slicing at the end of a program, it does not cater for slicing at arbitrary points in the program (middle slicing). Following Cartwright and Felleisen, it also uses the lazy semantics, which may appear to be somewhat counter intuitive for imperative programming languages.

4 Projections Theory And Trajectories

The projection theory [19,20] of Harman et al. was first used to examine the differences and similarities between amorphous and syntax-preserving forms of slicing. This theory defines slicing in terms of an ordering and an equivalence relation. For programs p and q:

- Write $q < p$ if q can be obtained from p by deleting statements and if p terminates then so does q.
- Write $q \sim p$ if p and q are 'semantically equivalent.'

For details of a suitable relation and proof that this really does define a partial ordering see, e.g., [19,20].

Definition 4.1 ($(<, \sim)$-slice) *Given an ordering $<$ and semantic equivalence $\sim$, program q is an $(<, \sim)$-slice of program p if and only if $q < p$ and $q \sim p$.*

In this paper we give a new definition of semantic equivalence that can be applied to non-terminating programs. We prove that it is indeed an equivalence relation and then show our main result: That it is preserved by Weiser's and the PDG algorithm. Thus we have a semantic equivalence relation that applies to non-terminating programs and is preserved by the current most commonly used slicing algorithms.

4.1 Weiser's Static Slicing

The semantic property that static slicing respects is based upon the concept of a *trajectory*. The following definitions of *trajectory*, *state restriction*, and *projection* are similar to those used by Weiser [38].

Definition 4.2 (Trajectory) *A* trajectory *is a finite sequence of label, state pairs:*

$$(l_1, \sigma_1)(l_2, \sigma_2) \ldots (l_k, \sigma_k)$$

If a program gives rise to trajectory $(l_1, \sigma_1)(l_2, \sigma_2) \ldots (l_k, \sigma_k)$ this means that the ith statement that was executed was labelled l_i and the resulting state[1] *is σ_i.*

Definition 4.3 (Restriction of a state to a set of variables) *Given a state, σ and a set of variables V, $\sigma{\downarrow}V$ restricts σ so that it is defined only for variables*

[1] Originally, in Weiser's definition, σ_i represents the state *before* executing the instruction at label l_i.

in V:

$$(\sigma\downarrow V)x = \begin{cases} \sigma\, x & \text{if } x \in V, \\ \bot & \text{otherwise.} \end{cases}$$

Definition 4.4 *For a label l' and state σ the projection of the trajectory sequence element (l', σ) to the slicing criterion (V, l) is*

$$(l', \sigma)\Downarrow(V, l) = \begin{cases} (l', \sigma\downarrow V) & \text{if } l' = l, \\ \lambda & \text{otherwise,} \end{cases}$$

where λ denotes the empty string.

Definition 4.5 (Projection of a trajectory to a slicing criterion) *The projection of the trajectory $T = (l_1, \sigma_1)(l_2, \sigma_2) \ldots (l_k, \sigma_k)$ to the slicing criterion (V, l) is*

$$\mathrm{Proj}_{(V,l)}(T) = (l_1, \sigma_1)\Downarrow(V, l) \ldots (l_k, \sigma_k)\Downarrow(V, l)$$

Definition 4.6 (Weiser's backward static slice) *A slice s of a program p on a slicing criterion (V, l) is any executable program with the following two properties.*

(1) s can be obtained from p by deleting zero or more statements.
(2) Whenever p halts on an input state σ with a trajectory T then s also halts on input σ with trajectory T' where $\mathrm{Proj}_{(V,l)}(T) = \mathrm{Proj}_{(V,l)}(T')$.

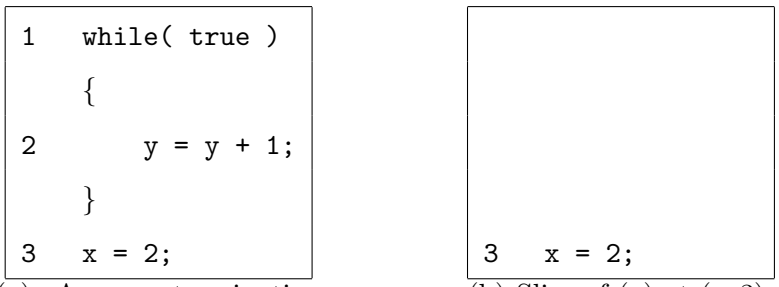
The trajectory $\mathrm{Proj}_{(V,l)}(T)$ is obtained first by deleting all elements of T whose label component is not l and then by restricting the state components to V.

Weiser himself noticed that there is no guarantee that a slice does not terminate whenever the original program does not terminate. Semantic equivalence for static backward slicing was previously [1,2,10,11] defined as follows:

Definition 4.7 (Static backward equivalence) *Given two programs p and q, and slicing criterion (V, l), p is* static backward equivalent *to q, written $p\ \mathcal{S}_{(V,l)}\ q$, if and only if for all input states σ, when the execution of p in σ gives rise to a trajectory T_p^σ and the execution of q in σ gives rise to a trajectory T_q^σ, then* $\mathrm{Proj}_{(V,l)}(T_p^\sigma) = \mathrm{Proj}_{(V,l)}(T_q^\sigma)$.

5 Static Backward Equivalence And Non-Termination

By Definition 4.2 the trajectories in Definition 4.7 are finite, hence static backward equivalence is only defined for programs which terminate. If $p\ \mathcal{S}_{(V,l)}\ q$ then p and q need only agree with respect to (V, l) in initial states where they



(a) A non-terminating program.

(b) Slice of (a) at $(\mathtt{x}, 3)$.

Fig. 6. A non-terminating program (a) (repeated from Figure 3(c)) and a slice of it (b) at $(\mathtt{x}, 3)$ using Weiser's algorithm. The slice terminates but the program does not. Program and slice are not semantically equivalent under Definition 4.7, even when admitting infinite trajectories: Program (a) has a single infinite trajectory that does not include line 3 whereas program (b) has a single finite trajectory including only line 3.

both terminate. Thus if there is no initial state in which p and q both terminate then vacuously $p \, \mathcal{S}_{(V,l)} \, q$. This means that $\mathcal{S}_{(V,l)}$ is not an equivalence relation on the set of all programs: It is only an equivalence relation on sets of programs which all terminate in the same set of initial states. (The set of programs which always terminate is an example of such a set).

In this section we consider how Weiser's algorithm slices programs which do not terminate in some initial states and how the definition of static backward equivalence could be extended to cover such cases.

5.1 Admitting Infinite Trajectories

An obvious extension is to admit infinite trajectories into Definition 4.7, and this works for the program and slice in Figure 2, i.e., using Weiser's algorithm gives a slice that preserves the modified semantics.

Unfortunately this simple extension does not generally work. A counter-example is given in Figure 6. The definition of semantic equivalence fails because the trajectories are no longer equal: Anything appearing in the program after an infinite loop does not appear in the trajectory, whereas it does appear in the trajectory of a slice in which the infinite loop has been removed.

5.2 Transfinite Trajectories

Trying to fix this problem leads us to consider trajectories which in the program in Figure 6(b) would include line 3. For example, we could say that the

trajectory of the program in Figure 6(b) is the infinite trajectory from the infinite loop *followed by* the assignment to `x` at line 3. Such *transfinite* trajectories were used by Giacobazzi and Mastroeni [18]. Now the infinite trajectory before line 3 will be ignored, leaving the final assignment to `x`.

The transfinite computation approach can fail when the program control flow after an infinite loop depends on values computed by an infinite loop: In Figure 5(a) the value of `x` after execution of the infinite loop is undefined. Therefore we do not know whether to take the true of the false branch of the `if`. Giacobazzi and Mastroeni [18] did not consider this.

To fix this problem we could consider this program to have a *set* of trajectories. In the case of the programs in Figure 5 each initial state would product *two* trajectories: One that takes the True branch of the `if` and the other taking the False branch. We would then require the program and its slice to agree on all of the trajectories in these sets.

5.3 Turtles All The Way Down

Using sets of trajectories to fix up the problem of values undetermined after an infinite loop leads to more problems. For the program in Figure 7(a) the set of trajectories we must consider consists of the infinite trajectory from the first loop followed by all possible finite and infinite length trajectories from the second loop. This set of trajectories is infinite.

```
1  while( true ) {
2    y = y + 1;
   }
3  while( y != z) {
4    y = y - 1;
   }

5  x = 2;
```

(a)

```
1  while( true ) {
2    y = y + 1;


3    while( y!= z ) {
4       y = y - 1;
     }
   }
5  x = 2;
```

(b)

Fig. 7. The transfinite trajectories approach with its sets of trajectories becomes very complex in the presence of multiple loops as the sets become infinite: In program (a) the second loop may execute any finite or infinite number of times. In program (b) the inner loop may execute any number of times on each iteration of the outer loop.

Matters are even more complex for the program in Figure 7(b) where we have

a loop within a loop. As the values of `x` and `y` are not known *a priori* the inner loop may execute any number of times – including infinitely – on each iteration of the outer loop.

We will show that there is a simpler solution that removes the need to consider transfinite trajectories and all their complications. We will define an intuitive equivalence on programs that does not require non-standard semantics or transfinite sequences in order to capture control and data dependence-based slicing algorithms.

6 Trajectory Semantics

We introduce a new equivalence over *all* programs (not just terminating ones) which both extends the previously defined mathematical framework (in the sense that restricting them both to the set of programs that always terminate yields identical relations) and is preserved by slicing algorithms based on data and control dependence.

Before we discuss trajectories, we first define our simple language with labels, sub-programs, and quotient programs. We do so using the following syntax:

$$
\begin{array}{rcl}
\Gamma & ::= & l : \texttt{skip} \mid \\
& & l : x{=}e \mid \\
& & \Gamma_1 ; \Gamma_2 \mid \\
& & \texttt{if } (l : B) \ \ \Gamma_1 \ \texttt{else } \Gamma_2 \mid \\
& & \texttt{while } (l : B) \ \ \Gamma
\end{array}
$$

where l denotes a label, e denotes an arithmetic expression and B denotes a boolean expression. We call a string in this language a *program*, and identify Γ with the set of all programs. Each component in the above disjunction is called a *sub-program*, and furthermore the concatenation and `if` components each contain two further sub-programs, and the `while` component contains one further sub-program.

If n is a sub-program of p then if we replace it by `skip` then we obtain a *quotient program* q of p by n. (If n is a component of the above disjunction then we may, equivalently, simply delete it instead.) The sub-programs of q that are also sub-programs of p are said to *survive*.

We define trajectory semantics denotationally. Denotational semantics [32,33], enables mathematical meaning to be given to programming languages. In

standard denotational semantics a state, $\sigma \in \Sigma$, is a mapping from the set $\mathbb{V}$ of program variables to the set V of values. For example, the function $\sigma = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ is the state where the value of x is 1, the value of y is 2 and the value of z is 3. Conventionally, an extra state $\perp$ is added to represent non-termination. So we get:

$$\Sigma_\perp = \Sigma \cup \{\perp\} = [\mathbb{V} \mapsto V] \cup \{\perp\}.$$

6.1 Standard Trajectory Semantics

Standard trajectory semantics, $\mathcal{T}$, is a map that takes a program and a state to a trajectory, i.e.,

$$\mathcal{T} : \Gamma \to \Sigma \to \text{Seq}(L \times \Sigma)$$

where Γ is the set of all programs and $\text{Seq}(L \times \Sigma)$ means the set of all finite and infinite sequences of label, state pairs.

In standard trajectory semantics we allow trajectories to be countably infinite. If p is a non-terminating program in state σ, then the trajectory of all programs of the form pq (where q is an arbitrary program) will be equal to the trajectory of p in state σ. We now give rules which define $\mathcal{T}$ for each syntactic category:

- For `skip` statements:
$$\mathcal{T}[\![l : \texttt{skip}]\!]\sigma = \langle (l, \sigma) \rangle$$
where l is the label and $\langle (l, \sigma) \rangle$ represents the singleton sequence consisting of the pair (l, σ).
- For assignment statements:
$$\mathcal{T}[\![l : x{=}e]\!]\sigma = \langle (l, \sigma[x \leftarrow \mathcal{E}[\![e]\!]\sigma]) \rangle$$
where $\mathcal{E}[\![e]\!]\sigma$ means the 'new' value resulting from evaluating expression e in state σ and $\sigma[x \leftarrow \mathcal{E}[\![e]\!]\sigma]$ is the state σ 'updated' with the maplet that takes variable `x` to this new value.
- For sequences of statements:
$$\mathcal{T}[\![p; q]\!]\sigma = \mathcal{T}[\![p]\!]\sigma \oplus \mathcal{T}[\![q]\!]\sigma'$$
where σ' is the state obtained [2] after executing p in σ and $\oplus$ means concatenation. Concatenating to the right of an infinite sequence has no ef-

[2] Note that $\sigma' = \mathcal{M}[\![p]\!]\sigma$ where $\mathcal{M} : \Gamma \to \Sigma_\perp \to \Sigma_\perp$ is the standard denotational meaning and can be defined in terms of $\mathcal{T}$ as follows:

$$\mathcal{M}[\![p]\!]\sigma = \begin{cases} \text{The state component of the last element of } \mathcal{T}[\![p]\!]\sigma & \text{if } \mathcal{T}[\![p]\!]\sigma \text{ is finite, or} \\ \perp & \text{otherwise.} \end{cases}$$

fect, i.e., if a is infinite then $a \oplus b = a$, and if A and B are sets then $A \oplus B = \{a \oplus b \mid a \in A \text{ and } b \in B\}$.

- For `if` statements:

$$\mathcal{T}[\![\texttt{if}(l : B)\ \ p\ \texttt{else}\ q]\!]\sigma = \langle (l, \sigma)\rangle \oplus (\mathcal{E}[\![B]\!]\sigma \to \mathcal{T}[\![p]\!]\sigma, \mathcal{T}[\![q]\!]\sigma)$$

  where $(\mathit{True} \to a, b)$ is a, $(\mathit{False} \to a, b)$ is b, and $(\bot \to a, b)$ is the empty sequence. In other words, for an `if` statement, the first element of the trajectory is the label of the `if` in the current state. The rest of the trajectory is the trajectory of one of the branches depending on the value of the boolean expression evaluated in the current state.

- For `while` loops:

$$\mathcal{T}[\![\texttt{while}(l : B)\ p]\!]\sigma = \mathcal{T}[\![\texttt{if}(l : B)\ \{p; \texttt{while}(l : B)p\}\ \texttt{else skip}]\!]\sigma$$

  that is, `while` loops are defined simply in terms of `if` statements in the standard way. Alternatively, one may prefer

$$\mathcal{T}[\![\ \texttt{while}(l : B)\ p]\!]\sigma = \langle l, \sigma\rangle \oplus \begin{cases} \mathcal{T}[\![p]\!]\sigma \oplus \mathcal{T}[\![\texttt{while}(l : B)\ p]\!](\mathcal{M}[\![p]\!]\sigma) & \text{if } \mathcal{E}[\![b]\!]\sigma, \text{ or} \\ \langle\rangle & \text{otherwise.} \end{cases}$$

Note that if $[\![\texttt{while}(l : B)\ p]\!]$ does not terminate in state σ then the sequence defined by $\mathcal{T}[\![\texttt{while}(l : B)\ p]\!]\sigma$ will be countably infinite.

For example, each of the three programs p in Figure 3 has trajectory

$$\mathcal{T}[\![p]\!] = \langle (1, \sigma)(2, \sigma[x \leftarrow x + 1]) \\ (1, \sigma')(2, \sigma'[x \leftarrow x + 1])(1, \sigma'')(2, \sigma''[x \leftarrow x + 1]) \ldots\rangle$$

for all initial states σ. The trajectories through the program in Figure 7(b) depend on the initial state. Putting $(x, y, z) = (1, 1, 0)$ in initial state σ gives the following trajectory:

$$\mathcal{T}[\![p]\!] = \langle (1, \sigma)(2, \sigma[y \leftarrow y + 1]) \\ (3, \sigma')(4, \sigma'[y \leftarrow y - 1])(3, \sigma)(4, \sigma[y \leftarrow y - 1])(3, \sigma'')(1, \sigma'') \ldots\rangle$$

where σ' is the state where $(x, y, z) = (1, 2, 0)$ and σ'' is the state where $(x, y, z) = (1, 0, 0)$.

6.2 Finite Trajectory Semantics

Our new finite trajectory semantics, $\vec{\mathcal{T}}$ is a function of type:

$$\vec{\mathcal{T}} : \mathbb{N} \to \Gamma \to \Sigma \to \mathrm{Seq}(L \times \Sigma)$$

In essence, it can be thought of as a sequence of semantic functions each of the same type as $\mathcal{T}$ defined in the previous section. The difference being that $\vec{\mathcal{T}}_n[\![p]\!]\sigma$ will be finite for all n, p, σ. This means we do not need to consider $\perp$. Informally, we can imagine that $\vec{\mathcal{T}}_n[\![p]\!]\sigma$ defines the trajectory of program p in state σ where all loops are allowed to iterate at most n times after which they are forced to terminate.

To define $\vec{\mathcal{T}}$ we need to unfold `while` loops to a finite number of iterations. We do this by simply replacing a `while` loop by n nested `if` statements containing the loop body. Note that in the case of nested loops it does not remove the loops from the body.

Definition 6.1 (Loop unfolding) *Let l be a label, B be a boolean expression, and $p \in \Gamma$. The nth unfolding of the* `while` *loop*

$$\texttt{while } (l : B)\ p$$

is defined inductively as follows:

$$W_0(l, B, p) = \texttt{if } (l : B)\ \ \texttt{skip else skip}$$
$$W_{n+1}(l, B, p) = \texttt{if } (l : B)\ \ p;\ W_n(l, B, p) \texttt{ else skip}$$

As before we now give rules which define $\vec{\mathcal{T}}$ for each syntactic category:

- For `skip` statements:
$$\vec{\mathcal{T}}_n[\![l : \texttt{skip}]\!]\sigma = \langle (l, \sigma) \rangle$$
- For assignment statements:
$$\vec{\mathcal{T}}_n[\![l : x{=}e]\!]\sigma = \langle (l, \sigma[x \ \leftarrow \mathcal{E}[\![e]\!]\sigma]) \rangle$$
- For sequences of statements:
$$\vec{\mathcal{T}}_n[\![p; q]\!]\sigma = \vec{\mathcal{T}}_n[\![p]\!]\sigma \oplus \vec{\mathcal{T}}_n[\![q]\!]\sigma'$$
where as before σ' is the state component of the last element of $\vec{\mathcal{T}}_n[\![p]\!]\sigma$.
- For `if` statements:
$$\vec{\mathcal{T}}_n[\![\texttt{if}(l : B)\ \ p \texttt{ else } q]\!]\sigma = \langle (l, \sigma) \rangle \oplus (\mathcal{E}[\![B]\!]\sigma \rightarrow \vec{\mathcal{T}}_n[\![p]\!]\sigma, \vec{\mathcal{T}}_n[\![q]\!]\sigma)$$
- For `while` loops:
$$\vec{\mathcal{T}}_n[\![\texttt{while } (l : B)\ \ p]\!]\sigma = \vec{\mathcal{T}}_n[\![W_n(l, B, p)]\!]\sigma$$
Alternatively, the `while` loop rule can be defined without unfolding as:
$$\vec{\mathcal{T}}_n[\![\texttt{while}(l : B)\ p]\!]\sigma = \vec{\mathcal{T}}^1_n[\![\texttt{while}(l : B)\ p]\!]\sigma$$

where

$$\vec{\mathcal{T}}_n^i[\![\texttt{while}(l : B)\ p]\!]\sigma = \langle l, \sigma\rangle \oplus \begin{cases} \vec{\mathcal{T}}_n[\![p]\!]\sigma \oplus \vec{\mathcal{T}}_n^{i+1}[\![\texttt{while}(l : B)\ p]\!]\sigma' & \text{if } \mathcal{E}[\![B]\!]\sigma \text{ and } i \leqslant n, \text{ or} \\ \langle\rangle & \text{otherwise.} \end{cases}$$

where, again, σ' is the state component of the last element of $\vec{\mathcal{T}}_n[\![S]\!]\sigma$.

For example the the program in Figure 3(a) we obtain the finite trajectories

$$\begin{aligned}
\vec{\mathcal{T}}_0[\![p]\!]\sigma &= \langle (1,\sigma)(3,\sigma[x \leftarrow 1])\rangle \\
\vec{\mathcal{T}}_1[\![p]\!]\sigma &= \langle (1,\sigma)(2,\sigma[x \leftarrow x+1])(1,\sigma')(3,\sigma'[x \leftarrow 1])\rangle \\
\vec{\mathcal{T}}_2[\![p]\!]\sigma &= \langle (1,\sigma)(2,\sigma[x \leftarrow x+1])(1,\sigma')(2,\sigma'[x \leftarrow x+1])(1,\sigma'')(3,\sigma''[x \leftarrow 1])\rangle \\
&\vdots
\end{aligned}$$

The trajectories through the program in Figure 7(b) depend on the initial state. Let σ_i be the state where $(x, y, z) = (1, i, 0)$ then:

$$\begin{aligned}
\vec{\mathcal{T}}_0[\![p]\!]\sigma_1 &= \langle (1,\sigma_1)(5,\sigma_1[x \leftarrow 2])\rangle \\
\vec{\mathcal{T}}_1[\![p]\!]\sigma_1 &= \langle (1,\sigma_1)(2,\sigma_1[y \leftarrow y+1])(3,\sigma_2)(4,\sigma_2[y \leftarrow y-1]) \\
&\qquad (3,\sigma_1)(1,\sigma_1)(5,\sigma_1[x \leftarrow 2])\rangle \\
\vec{\mathcal{T}}_2[\![p]\!]\sigma_1 &= \langle (1,\sigma_1)(2,\sigma_1[y \leftarrow y+1])(3,\sigma_2)(4,\sigma_2[y \leftarrow y-1]) \\
&\qquad (3,\sigma_1)(4,\sigma_1[y \leftarrow y-1])(3,\sigma_0) \\
&\qquad (1,\sigma_0)(2,\sigma_0[y \leftarrow y+1])(3,\sigma_1)(4,\sigma_0[y \leftarrow y-1]) \\
&\qquad (3,\sigma_0)(1,\sigma_0)(5,\sigma_0[x \leftarrow 2])\rangle \\
&\vdots
\end{aligned}$$

Notice that in $\vec{\mathcal{T}}_0[\![p]\!]\sigma_1$ the loop unfolding forces the outer loop to exit immediately. In $\vec{\mathcal{T}}_1[\![p]\!]\sigma_1$ both loops are forced to exit early. However in $\vec{\mathcal{T}}_n[\![p]\!]\sigma_1$ for $n \geqslant 2$ the inner loop exits naturally and only the outer loop is forced to exit.

7 Finite Trajectory Backward Slice Equivalence

Having defined $\vec{\mathcal{T}}$ we are now in a position to define the new equivalence relation $\vec{\mathcal{S}}$ between programs and their slices. Informally, a program p and its slice s are *finite trajectory backward slice equivalent* with respect to slicing criterion (V, l) if for all states σ and all $n \in \mathbb{N}$ the n-trajectory of p in σ and the n-trajectory of s in σ 'agree' at (V, l).

Definition 7.1 (Finite trajectory backward slice equivalence) *Let $p, q \in \Gamma$ be programs, and let (V, l) be a slicing criterion for p. Program p is* static backward equivalent *to q if and only if*

$$\forall \sigma \in \Sigma : \forall n \in \mathbb{N} : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_n[\![p]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_n[\![q]\!]\sigma).$$

We write $p \; \vec{\mathcal{S}}_{(V,l)} \; q$.

Lemma 7.1 *If (V, l) is a slicing criterion then $\vec{\mathcal{S}}_{(V,l)}$ is an equivalence relation.*

Proof. Let $p, q, r \in \Gamma$ be programs and (V, l) be a slicing criterion.

The relation is reflexive means that $p \; \vec{\mathcal{S}}_{(V,l)} \; p$ which is, by definition, equivalent to $\forall \sigma \in \Sigma : \forall m \in \mathbb{N} : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![p]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![p]\!]\sigma)$, which is trivially true.

For symmetry, suppose that $p \; \vec{\mathcal{S}}_{(V,l)} \; q$ which is, by definition, equivalent to

$$\forall \sigma \in \Sigma : \forall m \in \mathbb{N} : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![p]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![q]\!]\sigma).$$

This is equivalent to $\forall \sigma \in \Sigma : \forall m \in \mathbb{N} : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![q]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![p]\!]\sigma)$, which is $q \; \vec{\mathcal{S}}_{(V,l)} \; p$.

For transitivity, suppose that $p \; \vec{\mathcal{S}}_{(V,l)} \; q$ and $q \; \vec{\mathcal{S}}_{(V,l)} \; r$, i.e.,

$$\begin{aligned} \forall \sigma \in \Sigma : \forall m \in \mathbb{N} : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![p]\!]\sigma) &= \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![q]\!]\sigma) \\ \text{and } \forall \sigma \in \Sigma : \forall m \in \mathbb{N} : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![q]\!]\sigma) &= \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![r]\!]\sigma) \end{aligned}$$

which are equivalent to

$$\begin{aligned} \forall \sigma \in \Sigma : \forall m \in \mathbb{N} : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![p]\!]\sigma) &= \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![q]\!]\sigma) \\ \forall m \in \mathbb{N} : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![q]\!]\sigma) &= \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![r]\!]\sigma) \end{aligned}$$

from which it follows that $\forall \sigma \in \Sigma$ and $\forall m \in \mathbb{N}$

$$\mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![p]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![q]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![r]\!]\sigma). \qquad \square$$

Theorem 7.1 ($\vec{\mathcal{S}}$ is at least as accurate as $\mathcal{S}$) *For all slicing criteria (V, l) and all programs p and q, if $p \; \vec{\mathcal{S}}_{(V,l)} \; q$ then $p \; \mathcal{S}_{(V,l)} \; q$.*

Proof. Suppose that $p \; \vec{\mathcal{S}}_{(V,l)} \; q$ then

$$\forall \sigma \in \Sigma : \forall n \in \mathbb{N} : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_n[\![p]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_n[\![q]\!]\sigma).$$

As explained in the first paragraph of Section 5, we only need to consider the states σ in which p and q terminate.

Let m be the maximum number of times round all loops in p and q when starting in state σ. Then

$$\mathcal{T}[\![p]\!]\sigma = \vec{\mathcal{T}}_m[\![p]\!]\sigma \quad \text{and} \quad \mathcal{T}[\![q]\!]\sigma = \vec{\mathcal{T}}_m[\![q]\!]\sigma$$

and hence

$$\text{Proj}_{(V,l)}(\mathcal{T}[\![p]\!]\sigma) = \text{Proj}_{(V,l)}(\mathcal{T}[\![q]\!]\sigma). \qquad \square$$

Thus the new equivalence $\vec{\mathcal{S}}$ is at least as accurate as the old equivalence, $\mathcal{S}$. It is in fact much more accurate. For example, $\mathcal{S}$ cannot distinguish between programs that never terminate but as the examples in Section 5 show, $\vec{\mathcal{S}}$ can.

Theorem 7.2 ($\vec{\mathcal{T}}$ is substitutive) *Let p be a program and p' be the program obtained from p by replacing a sub-program q by program q'. Then $\forall n \in \mathbb{N}$*

$$\vec{\mathcal{T}}_n[\![q]\!] = \vec{\mathcal{T}}_n[\![q']\!] \implies \vec{\mathcal{T}}_n[\![p]\!] = \vec{\mathcal{T}}_n[\![p']\!].$$

Proof. We proceed by structural induction over our language Γ.

- For assignment statements and for `skip` statements the result is trivial.
- For sequences suppose the result holds for programs p_1 and p_2. Let p_1' be the program obtained by replacing a sub-program q_1 of p_1 by an equivalent program q_1'. Let p_2' be the program obtained by replacing a sub-program q_2, of p_2 by an equivalent program q_2'. For any $\sigma \in \Sigma$ and $n \in \mathbb{N}$ we have

$$\begin{aligned}
\vec{\mathcal{T}}_n[\![p_1'; p_2']\!]\sigma &= \vec{\mathcal{T}}_n[\![p_1']\!]\sigma \oplus \vec{\mathcal{T}}_n[\![p_2']\!]\sigma' \quad \text{(by definition)} \\
&= \vec{\mathcal{T}}_n[\![p_1]\!]\sigma \oplus \vec{\mathcal{T}}_n[\![p_2]\!]\sigma' \quad \text{(by induction)} \\
&= \vec{\mathcal{T}}_n[\![p_1; p_2]\!]\sigma
\end{aligned}$$

  from which the result follows by induction.
- For `if` statements let p_1, p_2, q_1, q_2, p_1', and p_2' be as above. For any $\sigma \in \Sigma$ and $n \in \mathbb{N}$ we have

$$\begin{aligned}
&\vec{\mathcal{T}}_n[\![\texttt{if}(l : B)\ \ p_1' \ \texttt{else}\ p_2']\!]\sigma \\
&\quad = \langle (l, \sigma) \rangle \oplus (\mathcal{E}[\![B]\!]\sigma \to \vec{\mathcal{T}}_n[\![p_1']\!]\sigma, \vec{\mathcal{T}}_n[\![p_2']\!]\sigma) \quad \text{(by definition)} \\
&\quad = \langle (l, \sigma) \rangle \oplus (\mathcal{E}[\![B]\!]\sigma \to \vec{\mathcal{T}}_n[\![p_1]\!]\sigma, \vec{\mathcal{T}}_n[\![p_2]\!]\sigma) \quad \text{(by induction)} \\
&\quad = \vec{\mathcal{T}}_n[\![\texttt{if}(l : B)\ \ p_1 \ \texttt{else}\ p_2]\!]\sigma
\end{aligned}$$

- For `while` loops suppose that the result holds for a program p and let p' be the program obtained by replacing a sub-program, q, of p by an equivalent program q'. For any $\sigma \in \Sigma$ and $n \in \mathbb{N}$ we have

$$\vec{\mathcal{T}}_n[\![\texttt{while}(l : B)\ p']\!]\sigma = \vec{\mathcal{T}}_n[\![W_n(l, B, p')]\!]\sigma$$

from which the result follows by induction using the result for `if` sub-programs and sequences of sub-programs. □

8 Control and Data Dependence Slicing Algorithms Satisfy Finite Trajectory Backward Slice Equivalence

Consider algorithms where the resulting slice consists of those statements which can be reached via control and data dependences from the slicing criterion (V, l). This includes Weiser's algorithm and the PDG algorithm which are the most widely used slicing algorithms. In our language (given in Section 6) control dependence can be defined in terms of the program structure.

Definition 8.1 (Control dependence) *For any* `if` *and* `while` *statements in a program p,*

$$\begin{aligned} \texttt{if}(l : B)\ q\ \texttt{else}\ r \quad &\Rightarrow \quad \forall l' \in \mathcal{L}(q) \cup \mathcal{L}(r) : l \xrightarrow[p]{\text{control}} l' \\ \texttt{while}(l : B)\ q \quad &\Rightarrow \quad \forall l' \in \mathcal{L}(q) : l \xrightarrow[p]{\text{control}} l' \end{aligned}$$

where $\mathcal{L}(q)$ denotes the labels in program q.

Data dependence in our language can be defined using finite syntactic paths.

Definition 8.2 (Finite syntactic paths) *For any program p, the set of finite syntactic paths of p, denoted by $\mathcal{P}(p)$, is defined as follows:*

$$\begin{aligned} \mathcal{P}([\![l : \texttt{skip}]\!]) &= \{l\} \\ \mathcal{P}([\![l : v := e]\!]) &= \{l\} \\ \mathcal{P}([\![q; r]\!]) &= \mathcal{P}(q) \oplus \mathcal{P}(r) \\ \mathcal{P}([\![\texttt{if}(l : B)\ q\ \texttt{else}\ r]\!]) &= \{l\} \oplus (\mathcal{P}(q) \cup \mathcal{P}(r)) \\ \mathcal{P}([\![\texttt{while}(l : B)\ q]\!]) &= (\{l\} \oplus \mathcal{P}(q))^* \oplus \{l\} \end{aligned}$$

Finite syntactic paths are similar to trajectories with the important difference that they do not contain state components and, as such, they do not take the actual value of the predicates into account, thereby allowing any branches to be taken. Therefore for any possible trajectory there is a corresponding finite syntactic path. We can now define data dependence.

Definition 8.3 (Data dependence) *For any program p data dependence is defined as follows: $l \xrightarrow[p]{\text{data}} l'$ if and only if $\exists \pi_1 l \pi_2 l' \pi_3 \in \mathcal{P}(p)$ with $\emptyset \neq \text{def}(l) \subseteq \text{ref}(l')$ and for all $l'' \in \pi_2$, $\text{def}(l'') \neq \text{def}(l)$. Each π_i may be an empty path,*

$\mathrm{def}(l)$ *is the set of variables defined at* l *(a set of maximum one element), and* $\mathrm{ref}(l)$ *denotes the set of variables referenced at* l*.*

Recall from Section 6 that deleting sub-programs (or replacing them by `skip`) from a program p leaves behind a quotient program q, and the sub-programs left in q are said to 'survive'. That is, the current dependence based slicing algorithms preserve our semantics. This is Theorem 8.1 and is our main result.

Now we are able to define what property the dependence-based slicing algorithms (including Weiser's one) preserve.

Definition 8.4 (Dependence-based slice) *Let* q *be a quotient of* p*, then* q *is a dependence-based slice of* p *with respect to* (V, l) *if and only if*

- l *survives in* q*, and*
- *if* l' *survives in* q *and either* $l'' \xrightarrow[p]{\text{data}} l'$ *or* $l'' \xrightarrow[p]{\text{control}} l'$ *then* l'' *survives in* q*.*

where l *is the label of an assignment sub-program and* $V = \mathrm{def}(l)$*.*

Note that in the above definition we allow slicing only with respect to the set of variables defined at the slice point. This is a very natural restriction and avoids the problem of the slice-point not being in the slice [3]. In the PDG approach [24] where there is a desire to slice on a criterion (V, l) that does not satisfy this a new node/assignment is added at the point of interest. This converts the problem of producing the slice for (V, l) into producing the slice for (V', l') for a (V', l') that does satisfy the above constraint.

Our goal is to show that for any program p, if q denotes the dependence-based slice of p with respect to (V, l) conforming to Definition 8.4 (the slice produced by Weiser's and the PDG algorithm) then $p\ \vec{\mathcal{S}}_{(V,l)}\ q$.

We generalise Definition 4.4 to a set of labels L by putting

$$(l', \sigma)\!\downarrow\! L = \begin{cases} (l, \sigma\!\downarrow\!\mathrm{def}(l)) & \text{if } l \in L, \\ \lambda & \text{otherwise.} \end{cases}$$

where we have taken $V = \mathrm{def}(l)$ for each $l \in L$. Similarly, we generalise Definition 4.5 to give

$$\mathrm{Proj}_L(T) = (l_1, \sigma_1)\!\downarrow\! L \ldots (l_k, \sigma_k)\!\downarrow\! L$$

for the trajectory $T = (l_1, \sigma_1)(l_2, \sigma_2) \ldots (l_k, \sigma_k)$.

[3] Weiser handled this problem by slicing with respect to the 'nearest successor' of the slice point that is in the slice [37].

Lemma 8.1 *Let q be a dependence-based slice of p with respect to (V, l) where l is an assignment statement and $V = \mathrm{def}(l)$, then*

$$\forall \sigma \in \Sigma : \forall m \in \mathbb{N} : \mathrm{Proj}_L(\vec{\mathcal{T}}_m[\![q]\!]\sigma) = \mathrm{Proj}_L(\vec{\mathcal{T}}_m[\![p]\!]\sigma)$$

where L is the set of surviving labels of q.

Proof. Write

$$\begin{aligned} \vec{\mathcal{T}}_m[\![p]\!]\sigma &= (l_1, \sigma_1) \ldots (l_j, \sigma_j) \\ \text{and } \vec{\mathcal{T}}_m[\![q]\!]\sigma &= (l'_1, \sigma'_1) \ldots (l'_k, \sigma'_k) \end{aligned}$$

where $\sigma = \sigma_0 = \sigma'_0$ so that

$$\mathrm{Proj}_L(\vec{\mathcal{T}}_m[\![p]\!]\sigma) = (l_{f(1)}, \sigma_{f(1)}{\downarrow}\mathrm{def}(l_{f(1)})) \ldots (l_{f(i)}, \sigma_{f(i)}{\downarrow}\mathrm{def}(l_{f(i)})) \quad (1)$$

$$\text{and } \mathrm{Proj}_L(\vec{\mathcal{T}}_m[\![q]\!]\sigma) = (l'_{g(1)}, \sigma'_{g(1)}{\downarrow}\mathrm{def}(l'_{g(1)})) \ldots (l'_{g(h)}, \sigma'_{g(h)}{\downarrow}\mathrm{def}(l'_{g(h)})) \quad (2)$$

where f and g are monotonic increasing functions on $\mathbb{N}$ and i and h are the lengths of the projected trajectories.

Suppose that the projected trajectories are not equal, then this is because at least one of the following is true:

(1) The projected trajectories do not have the same length, i.e., $h \neq i$.
(2) The projected trajectories have the same length, but the labels do not correspond, i.e., $h = i$ but $\exists t$ for which $l_{f(t)} \neq l'_{g(t)}$.
(3) The projected trajectories have the same length and the labels correspond, but the states differ at some point, i.e., $h = i$ and $\forall t\ l_{f(t)} = l'_{g(t)}$, but $\exists t$ and $\exists x \in \mathrm{def}(l_{f(t)})$ for which $\sigma_{f(t)}(x) \neq \sigma'_{g(t)}(x)$.

Taking the third case first, let t be minimal, i.e., the first position in which the projected states differ. As $\mathrm{def}(l_{f(t)}) \neq \emptyset$ this is an assignment and to cause the disagreement we must have $v \in \mathrm{ref}(l_{f(t)})$ with $\sigma_{f(t)-1}(v) \neq \sigma'_{g(t)-1}(v)$. But the minimality of t means that the projected trajectories (1) and (2) are equal up to their t-th elements, therefore to cause the disagreement on v there must be an assignment to v in p from a statement labelled $l'' \notin L$. Therefore $l'' \xrightarrow[p]{\mathrm{data}} l_{f(t)}$ which contradicts its absence from q and thus from $\mathrm{Proj}_L(\vec{\mathcal{T}}_m[\![q]\!]\sigma)$.

In the other two cases we again have agreement on some initial segment up to $l_{f(t)} = l'_{g(t)}$ for minimal t. Since the projected trajectories (1) and (2) now diverge $l_{f(t)}$ must label the entry point of an `if` or `while` sub-program and the boolean expression evaluated in p must differ to that evaluated in q. Thus as before we must have $v \in \mathrm{ref}(l_{f(t)})$ such that $\sigma_{f(t)}(v) \neq \sigma'_{g(t)}(v)$. However, arguing as for the third case, the states must also agree up to $\sigma_{f(t)} = \sigma'_{g(t)}$ and if v is assigned in p then its label is in L, giving the required contradiction.□

Theorem 8.1 *Dependence-based slicing algorithms satisfy the new equivalence. That is, if q is a dependence-based slice of p with respect to (V, l), where l is an assignment statement and $V = \mathrm{def}(l)$, then $q \; \vec{\mathcal{S}}_{(V,l)} \; p$.*

Proof. That $\forall \sigma \in \Sigma : \forall n \in \mathbb{N} : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_n[\![q]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_n[\![p]\!]\sigma)$ follows immediately from Lemma 8.1. □

9 A Comparison with the Infinite Trace Approaches

The preservation of our semantics by Weiser's algorithm is apparent in Figure 2. Take for example the trajectory $\vec{\mathcal{T}}_1[\![p]\!]\sigma$ that enters both `if` sub-programs:

$$\vec{\mathcal{T}}_1[\![p]\!]\sigma = (1,\sigma)(2,\sigma[x \leftarrow \texttt{getTemp()}])(3,\sigma'[y \leftarrow \texttt{getPressure()}]) \\ (4,\sigma'')(5,\sigma'')(6,\sigma'')(7,\sigma'')(1,\sigma'')$$

We wish to slice on line 7. Taking the closure under control and data dependence as in Definition 8.4 we obtain $L = \{1, 2, 6, 7\}$ and the projection of this trajectory to the slicing criterion $(\{x\}, L)$ is:

$$\mathrm{Proj}_{(\{x\},L)}(\vec{\mathcal{T}}_1[\![p]\!]\sigma) = (1,\sigma\!\downarrow\!L)(2,\sigma[x \leftarrow \texttt{getTemp()}]\!\downarrow\!L) \\ (6,\sigma'\!\downarrow\!L)(7,\sigma'\!\downarrow\!L)(1,\sigma'\!\downarrow\!L)$$

This agrees with Figure 2(b). This is true for all other finite trajectories $\vec{\mathcal{T}}_n[\![p]\!]\sigma$ for $n \in \mathbb{N}$.

Similarly, the preservation of our semantics for the program in Figure 6(a) and Weiser's slice in Figure 6(b) is straightforward.

For the program in Figure 3(b) the final value of `x` is undefined. Thus our semantics, which extends standard semantics, agrees with the non-standard transfinite computation approaches in this case.

The transfinite computation approaches became more problematic for the programs in Figure 5 because they consider these programs to be inequivalent. As we proved in Theorem 7.2 our semantics is substitutive and thus considers the programs in Figure 5 to be equivalent. For our finite trajectory semantics the trajectory $\vec{\mathcal{T}}_n[\![p]\!]\sigma$ includes line 6 if and only if n is even and line 8 if and only if n is odd.

For the programs in Figure 7 the final value of `y` is undefined, but as the finite trajectories we listed on page 17 show, the final value of `x` is clearly always 2.

Rather more subtly, in infinite and transfinite trajectory approaches the final value of a variable after executing a non-terminating loop will be the limit

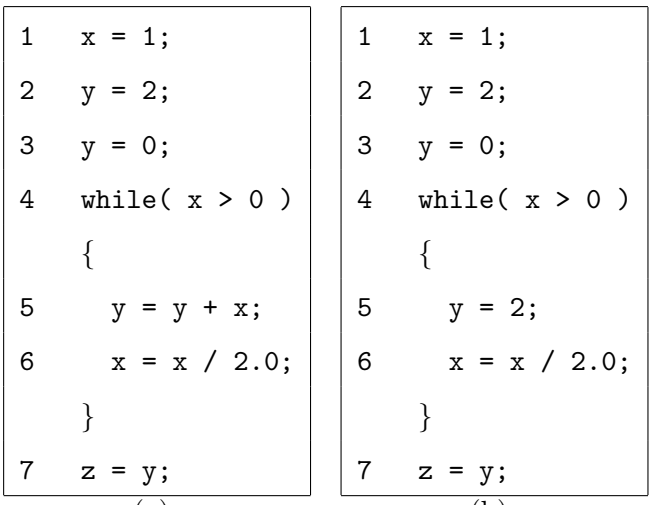


Fig. 8. In (a) the value of y converges to 2 whereas in (b) it is 2. These programs are equivalent under the transfinite trajectory semantics of Giacobazzi and Mastroeni [18], but not under our semantics.

of its values after each iteration. For example the programs in Figures 8(a) and 8(b) are equivalent for the value of z at the label 7 under the transfinite trajectory semantics of Giacobazzi and Mastroeni [18]. This is not so under our finite trajectory semantics because on any finite trajectory

$$\begin{aligned}\vec{\mathcal{T}}_n[\![p]\!]\sigma = (1,\sigma_0[x \leftarrow 1])(2,\sigma_1[y \leftarrow 0])(3,\sigma_2)(4,\sigma_2[y \leftarrow y{+}x])(5,\sigma_3[x \leftarrow x/2]) \\ (3,\sigma_4)(4,\sigma_4[y \leftarrow y+x])(5,\sigma_5[x \leftarrow x/2])\ldots \\ \ldots(3,\sigma_{2n})(4,\sigma_{2n}[y \leftarrow y+x])(5,\sigma_{2n+1}[x \leftarrow x/2])(6,\sigma_{2n+2}[z \leftarrow y])\end{aligned}$$

through the program in Figure 8(a) the final value of y is never 2.

The equivalence imparted by the semantics of Giacobazzi and Mastroeni [18] differs from our semantics when values converge but do not stabilise. This situation is illustrated by the slice in Figure 9. Under the semantics of Giacobazzi and Mastroeni [18] the program in in Figure 9(b) is a slice for the value of z on line 7. The slice in Figure 9(c) is produced by Weiser's algorithm, and is equivalent to the original under our semantics. The slice in Figure 9(c) fits better with the normal understanding of slicing. In addition, normal applications of slicing do not consider convergence and thus we argue that the semantics given in this paper is more appropriate for slicing. Recall that one of the main motivations was the context of reactive systems that are not intended to terminate. Here the value of a variable converging to a required value but not reaching this could be seen as if taking an infinite time for the required value to be obtained. Clearly, we would want to distinguish between this and the required value being reached in finite time.

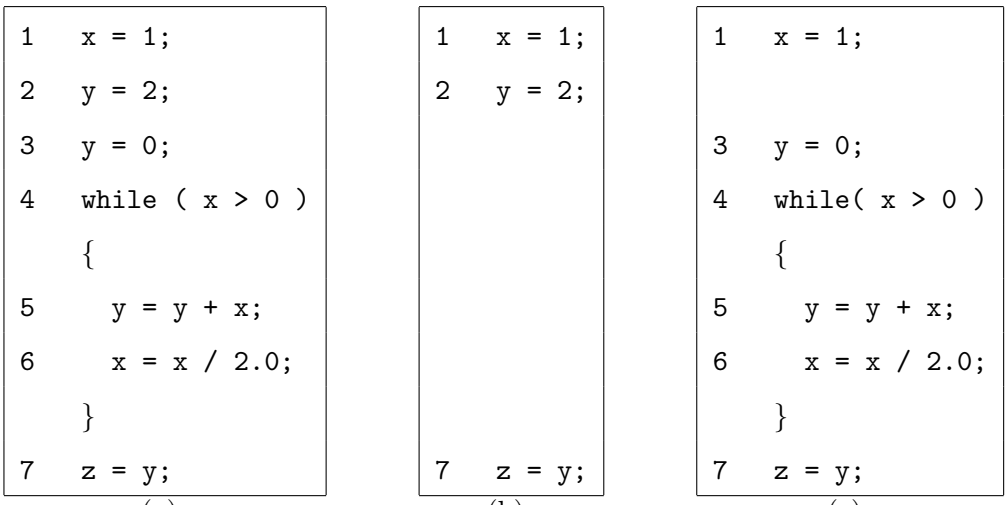


Fig. 9. Slicing on the value of `z` at label 7 produces the slice (b) when using transfinite semantics. Current slicing algorithms produce the slice in (c) which is equivalent to (a) under our semantics.

10 Conclusions and Future Work

In this paper we have defined a new semantics and associated equivalence of programs and showed that the equivalence is preserved by dependence based program slicing algorithms *including* for non-terminating programs. Our new finite trajectory semantics is based on standard denotational semantics. Our new equivalence stipulates that a non-terminating program and its slice must 'agree' in all terminating 'approximations'. We prove that this equivalence is preserved by Weiser's algorithm and other slicing algorithms which produce slices from the transitive closure of the data and control dependence relation applied to the slicing criterion. Our new equivalence also has useful properties such as substitutivity not possessed by the semantics of Cartwright and Felleisen [8] nor by the semantics of Giacobazzi and Mastroeni [18].

By contrast, related work uses non-standard semantics, and we saw how these semantics produce different slices to those produced by Weiser's and the PDG algorithm. Our approach is simpler and more intuitive than other attempts to define equivalence in non-terminating programs – especially compared to the transfinite computation approach [18] since, for example, we only consider finite traces and there is no need to consider infinite, let alone transfinite traces.

We see three main areas for future work: To generlise our work to program schemas, to apply the same ideas to other forms of slicing, and to investigate more accurate semantics.

In this paper we have always considered concrete programs, but it is well-

know that program schemas – equivalence classes of programs – contain all the necessary information to formally investigate dependence based slicing algorithms.

We focussed on static backward slicing, and especially on the extension of the static backward equivalence relation $\mathcal{S}$ on terminating programs to $\vec{\mathcal{S}}$ on all programs. Our idea of limiting loops to a specific number of iterations (or, equivalently, unfolding them for a specific number of times) can be applied to all other forms of slicing formally treated in our previous work [1,2,10,11]. Most importantly, it can be applied to the unified equivalence defined in [**?**]. We could also investigate the relationship between the resulting (dynamic) definitions and the existing (dynamic) slicing algorithms.

It is probably possible to generalise the semantics further, to allow other unfoldings apart from the ones considered here. This would lead to an even stronger equivalence.

11 Acknowledgements

The authors are very grateful to Roberto Giacobazzi, Isabella Mastroeni and Martin Ward for many detailed and thought–provoking discussions about the formalisation of program slicing.

This work is supported, in part, by EPSRC grant EP/E002919/1 (principal investigator Sebastian Danicic, co-investigators Mark Harman and Robert M. Hierons, and research assistant, Michael R. Laurence.).

Dave Binkley is supported by National Science Foundation grant CCR0305330.

Ákos Kiss is supported by The Péter Pázmány Program of the Hungarian National Office of Research and Technology (no. RET-07/2005).

Richard Barraclough is supported by KTP grant 1575 and @UK plc.

References

[1] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Ákos Kiss, B. Korel, Theoretical foundations of dynamic program slicing, Theoretical Computer Science 360 (1) (2006) 23–41.

[2] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, L. Ouarbya, Formalizing executable dynamic and forward slicing, in: 4^{th} International

Workshop on Source Code Analysis and Manipulation (SCAM 04), IEEE Computer Society Press, Los Alamitos, California, USA, 2004.

[3] D. W. Binkley, The application of program slicing to regression testing, Information and Software Technology 40 (11 and 12) (1998) 583–594.

[4] D. W. Binkley, K. B. Gallagher, Program slicing, in: M. Zelkowitz (ed.), Advances in Computing, Volume 43, Academic Press, 1996, pp. 1–50.

[5] D. W. Binkley, M. Harman, A survey of empirical results on program slicing, Advances in Computers 62 (2004) 105–178.

[6] D. W. Binkley, S. Horwitz, T. Reps, Program integration for languages with procedure calls, ACM Transactions on Software Engineering and Methodology 4 (1) (1995) 3–35.

[7] G. Canfora, A. Cimitile, A. De Lucia, Conditioned program slicing, Information and Software Technology Special Issue on Program Slicing 40 (11 and 12) (1998) 595–607.

[8] R. Cartwright, M. Felleisen, The semantics of program dependence, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, 1989.

[9] S. Danicic, , M. Harman, J. Howroyd, L. Ouarbya, A non-standard semantics for program slicing and dependence analysis, Logic and Algebraic Programming, Special Issue on Theory and Foundations of Programming Language Interference and Dependence 72 (2007) 123–240.

[10] S. Danicic, D. Binkley, T. Gyimóthy, M. Harman, Ákos Kiss, B. Korel, Minimal slicing and the relationships between forms of slicing, in: 5^{th} IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 05), IEEE Computer Society Press, Los Alamitos, California, USA, 2005, best paper award winner.

[11] S. Danicic, D. Binkley, T. Gyimóthy, M. Harman, Ákos Kiss, B. Korel, A formalisation of the relationship between forms of program slicing, Science of Computer Programming 62 (3) (2006) 228–252.

[12] S. Danicic, C. Fox, M. Harman, R. M. Hierons, J. Howroyd, M. Laurence, Slicing algorithms are minimal for programs which can be expressed as linear, free, liberal schemas, The Computer Journal 48 (6) (2005) 737–748.

[13] S. Danicic, M. Harman, R. Hierons, J. Howroyd, M. Laurence, Applications of linear program schematology in dependence analysis, in: $1^{st.}$ International Workshop on Programming Language Interference and Dependence, Verona, Italy, 2004.
URL `http://profs.sci.univr.it/ mastroen/noninterference.html`

[14] S. Danicic, M. Harman, J. Howroyd, L. Ouarbya, A lazy semantics for program slicing, in: $1^{st.}$ International Workshop on Programming Language Interference and Dependence, Verona, Italy, 2004.
URL `http://profs.sci.univr.it/ mastroen/noninterference.html`

[15] A. De Lucia, Program slicing: Methods and applications, in: 1^{st} IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, Los Alamitos, California, USA, 2001.

[16] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (3) (1987) 319–349.

[17] K. B. Gallagher, J. R. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering 17 (8) (1991) 751–761.

[18] R. Giacobazzi, I. Mastroeni, Non–standard semantics for program slicing, Higher-Order and Symbolic Computation 16 (4) (2003) 297–339, special issue on Partial Evalution and Semantics-Based Program Manipulation.

[19] M. Harman, D. W. Binkley, S. Danicic, Amorphous program slicing, Journal of Systems and Software 68 (1) (2003) 45–64.

[20] M. Harman, S. Danicic, Amorphous program slicing, in: 5^{th} IEEE International Workshop on Program Comprenhesion (IWPC'97), IEEE Computer Society Press, Los Alamitos, California, USA, 1997.

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

[22] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper, Testability transformation, IEEE Transactions on Software Engineering 30 (1) (2004) 3–16.

[23] J. Hatcliff, M. B. Dwyer, H. Zheng, Slicing software for model construction, Higher-Order and Symbolic Computation 13 (4) (2000) 315–353.

[24] S. Horwitz, T. Reps, D. W. Binkley, Interprocedural slicing using dependence graphs, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia, 1988, proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[25] S. Horwitz, T. Reps, D. W. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12 (1) (1990) 26–61.

[26] R. Komondoor, S. Horwitz, Semantics-preserving procedure extraction, in: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00), ACM Press, N.Y., 2000.

[27] B. Korel, J. Laski, Dynamic program slicing, Information Processing Letters 29 (3) (1988) 155–163.

[28] B. Korel, I. Singh, L. Tahat, B. Vaysburg, Slicing of state based models, in: IEEE International Conference on Software Maintenance (ICSM'03), IEEE Computer Society Press, Los Alamitos, California, USA, 2003.

[29] A. Lakhotia, P. Singh, Challenges in getting formal with viruses, virus bulletin.

[30] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, J. Hatcliff, A new foundation for control-dependence and slicing for modern program structures, in: European Symposium on Programming, 2005.

[31] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, M. B. Dwyer, A new foundation for control dependence and slicing for modern program structures, ACM Transactions on Programming Languages and Systems 29 (5).

[32] D. A. Schmidt, Denotational semantics: A Methodology for Language Development, Allyn and Bacon, 1986.

[33] J. E. Stoy, Denotational semantics: The Scott–Strachey approach to programming language theory, MIT Press, 1985, third edition.

[34] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (3) (1995) 121–189.

[35] M. Ward, Program slicing via FermaT transformations, in: 26^{th} IEEE Annual Computer Software and Applications Conference (COMPSAC 2002), IEEE Computer Society Press, Los Alamitos, California, USA, 2002.

[36] M. Ward, H. Zedan, Slicing as a program transformation, ACM Trans. Program. Lang. Syst. 29 (2) (2007) 7.

[37] M. Weiser, Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method, Ph.D. thesis, University of Michigan, Ann Arbor, MI (1979).

[38] M. Weiser, Program slicing, IEEE Transactions on Software Engineering 10 (4) (1984) 352–357.

[39] B. Xu, J. Qian, X. Zhang, Z. Wu, L. Chen, A brief survey of program slicing, ACM SIGSOFT Software Engineering Notes 30 (2) (2005) 1–36.
URL `http://doi.acm.org/10.1145/1050849.1050865`