A Trajectory-based Strict Semantics for Program Slicing

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

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

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

[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

We define a program semantics that is preserved by dependence-based slicing algorithms. It is a natural extension, to non-terminating programs, of the semantics introduced by Weiser (which only considered terminating ones) and, as such, is an accurate characterisation of the semantic relationship between a program and the slice produced by these algorithms.

Unlike other approaches, apart from Weiser's original one, it is based on strict standard semantics which models the 'normal' execution of programs on a von Neumann machine and, thus, has the advantage of being intuitive. This is essential since one of the main applications of slicing is program comprehension. Although our semantics handles non-termination, it is defined wholly in terms of finite trajectories, without having to resort to complex, counter-intuitive, non-standard models of computation. As well as being simpler, unlike other approaches to this problem, our semantics is substitutive. Substitutivity is an important property because it greatly enhances the ability to reason about correctness of meaning-preserving program transformations such as slicing.

1 Introduction

Program slicing is a program transformation technique for extracting an executable sub-program, the slice, from a larger program. The slice must preserve some semantic property of the program. Typically, this property is defined in terms of the values of a set of variables at a point of interest in the program.

Comprehensive surveys of program slicing, applications, techniques and results can be found in several papers [6,7,16,22,42,48]. Applications of slicing include program comprehension [20], software maintenance [18], testing and debugging [2,23], virus detection [33], integration [8], refactoring [30], restructuring, reverse engineering and reuse [9]. There are many forms of program slicing: static [46], dynamic [31] and conditioned [9]; forward and backward [26]; amorphous and syntax preserving [20]; non-termination removing [44,45] and non-termination preserving [37].

Historically, slicing was developed first as an algorithm; practice preceded theory. The most well-known and widely used slicing algorithms are variants of Weiser's Algorithm [46] and the PDG algorithm of Horwitz and Reps [25]. These algorithms are essentially the same and produce identical results because they are both based on the notions of data and control dependence [17].

Recent developments in the application of slicing to reactive programs [24] and finite state machine models [32] require slices of programs and models that may fail to terminate. This has led, for example, to new definitions of control flow for such non-terminating programs and algorithms for computing slices using them [36,37]. This interest in slicing non-terminating programs provides renewed impetus to 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.

Problems arise in trying to define the semantic relationship that slicing must preserve in the case of non-termination. Some slicing theories simply ignore non-terminating programs [1,3–5,45,46]. This is no longer acceptable since programs where non-termination is both normal and desirable – such as reactive systems – are increasingly common. As we demonstrate, existing slicing theory faces problems when non-termination is introduced, but the associated slicing algorithms often produce sensible slices. This suggests that the semantics of slicing non-terminating programs should be a natural extension of existing theory.

The primary contributions of this paper can be summarised as follows:

(1) We introduce a new, substitutive, intuitive, finite trajectory-based seman-

tics, $\vec{\mathcal{T}}$, of programs (Section 4.2). This semantics removes the need to consider complex, non-intuitive concepts such as transfinite computation, where programs are thought of as continuing to execute *after* finishing infinite loops.

(2) We prove (Theorem 4.1) that its induced equivalence, $\vec{\mathcal{S}}_{(V,l)}$, is a natural extension, to non-terminating programs, of the equivalence, $\mathcal{S}_{(V,l)}$, introduced by Weiser [46] that only considers terminating programs.

(3) We prove (Theorem 4.2) that slicing algorithms based on traditional data and control dependence preserve $\vec{\mathcal{T}}$, thereby showing that the new semantics captures the behaviour of program slicing algorithms for both terminating and non terminating programs.

In Section 2, we consider previous attempts to define the semantics that is preserved by dependence-based slicing algorithms. First we describe the finite trajectory semantics first introduced by Weiser [46] and later used by the authors [5,1,3,4]. We then consider alternative approaches which involve non-standard semantics. In particular, we consider the approaches of Cartwright and Felleisen [10], Giacobazzi and Mastroeni [19], and Nestra [35].

In Section 3, we illustrate the various shortcomings of previous approaches, including the lack of applicability of Weiser semantics to non-terminating programs and problems with the lazy and transfinite approaches including their lack of substitutivity, their undefined control, and their complex and counter-intuitive nature. These shortcomings motivate the need for a new semantics.

Our new semantics and equivalence is introduced in Section 4. In Subsection 4.2, we define $\vec{\mathcal{T}}$, and in Subsection 4.3, introduce finite trajectory backward slice equivalence, $\vec{\mathcal{S}}_{(V,l)}$, based on $\vec{\mathcal{T}}$ and prove that it is a natural extension, to non-terminating programs, of static backward equivalence first introduced by Weiser. In Subsection 4.4 we prove the main result – that dependence-based slicing algorithms produce slices which are equivalent to the original with respect to $\vec{\mathcal{S}}_{(V,l)}$.

In Section 5, we compare our new approach with past contributions, proving that unlike these approaches, it has the essential property of substitutivity (Theorem 5.1) and argue also that it is simpler and more intuitive. Finally, in Section 6 we conclude and consider areas for future work.

2 Background and Related Work

In this section, we briefly describe previous attempts to define the semantics that is preserved by dependence-based slicing algorithms. First we describe the finite trajectory semantics first introduced by Weiser [46] and later used by the authors [5,1,3,4]. Following this, we describe the non-standard semantics-based approaches of Cartwright and Felleisen [10], Giacobazzi and Mastroeni [19] and Nestra [35].

2.1 Weiser's Finite Trajectory Semantics

Before a program is sliced, a *slicing criterion* must be specified. A slicing criterion is a pair, (V, l) where V is a set of variables of interest and l is a label representing the program point of interest. The slicing algorithm will return a program which agrees with the program being sliced at the point of interest, l, with respect to the variables of interest V every time both the original program and the slice pass through l. This idea is expressed using trajectories [46,47]:

Definition 2.1 (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.*

Since a slice only needs to preserve the behaviour of the program with respect to the variables of interest, Weiser uses the concept of state restriction:

Definition 2.2 (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 in V:*

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

Since the programs need only to agree at the slicing criterion, Weiser introduced the idea of restricting a trajectory to a slicing criterion (V, l). To do this, first delete all pairs in the trajectory whose label component is not l and for the ones that are left restrict the state component of the pair to V as just defined.

First we define how to project a single element of a trajectory onto a slicing criterion:

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

Definition 2.3 *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 2.4 (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*

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

Having set up the semantics, Weiser then defines the property which captures the relationship between a program and its slice.

Definition 2.5 (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 $\mathrm{Proj}_{(V,l)}(T) = \mathrm{Proj}_{(V,l)}(T')$.

Binkley et al. restate this as a semantic equivalence, $\mathcal{S}_{(V,l)}$, for static backward slicing [5,1,3,4] as follows:

Definition 2.6 (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 $\mathrm{Proj}_{(V,l)}(T_p^\sigma) = \mathrm{Proj}_{(V,l)}(T_q^\sigma)$.*

Reps and Yang [38], in effect, show that slices produced by dependence-based algorithms such as Weiser's preserve static backward equivalence.

Despite the fact that Weiser makes no claims for the behaviour of his slicing algorithm in the presence of non-termination, several authors, including Weiser himself, noticed that slices produced from a non-terminating program by his algorithm behave in a consistent, compelling and meaningful manner. In Figure 1 for example, even though the program, $p_{1(a)}$ being sliced does not terminate, the behaviour of the slice $p_{1(b)}$ agrees with $p_{1(a)}$ at the slicing criterion *i.e.,* they both pass through the slicing criterion the same number of times and the values of x agree each time. Under standard strict semantics, however, the meaning of both these programs is simply *undefined*: the same as any other non-terminating program. (Even ones which do not agree at the

```
1  while( true ) {
2    x = getTemp();
3    y = getPressure();
4    if( y > pThreshold ) {
5       pressureAlarm();
     }
6    if( x > tThreshold ) {
7       checkpoint(x);
     }
   }
```

(a) A reactive program: $p_{1(a)}$

```
1  while( true ) {
2    x = getTemp();




6    if( x > tThreshold ) {
7       checkpoint(x);
     }
   }
```

(b) The Weiser slice, $p_{1(b)}$ at slicing criterion $(x, 7)$

Fig. 1. A reactive program $p_{1(a)}$ and Weiser's slice of it $p_{1(b)}$ Although $p_{1(a)}$ is very simple, it is typical of the kinds of infinite computation found in reactive systems. Weiser's slice, $p_{1(b)}$ with respect to slicing criterion $(x, 7)$ is 'sensible', even though Weiser's semantics only applies to terminating programs.

slicing criterion in this way.) This shows that standard strict semantics does not precisely capture the behaviour preserved by slicing algorithms.

2.2 Non-standard Semantic Approaches to Program Slicing

The observation that dependence-based slicing algorithms do not preserve standard semantics led several authors to investigate non-standard semantics in an attempt to capture more precisely the behaviour preserved by these algorithms.

2.2.1 The Lazy Semantics of Cartwright and Felleisen

Cartwright and Felleisen [10] were the first to introduce a non-standard semantics for program slicing. In this semantics, they allow partially defined states, i.e., ones where some variables are mapped to undefined ($\perp$), representing non-termination and others are mapped to proper values. The value of a variable becomes $\perp$ if it is assigned to inside an infinite loop. Programs, however, continue to execute after infinite loops and hence the value of a variable can 'recover' from being $\perp$. For example,in the program $p_{2(c)}$ in Figure 2, variable `x` has final value undefined, but variable `y` has final value 1. This shows that the semantics is not strict. Under standard strict semantics, of

```
while( true ) {
  x = x + 1;
}
x = 1;
```

(a) Program $p_{2(a)}$

```
while( true ) {
   x = x + 1;
}
```

(b) Program $p_{2(b)}$

```
while( true ) {
  x = x + 1;
}
y = 1;
```

(c) Program $p_{2(c)}$

Fig. 2. Both the lazy semantics of Cartwright and Felleisen and the transfinite computation semantics of Giacobazzi and Mastroeni give the final value of `x` to be 1 in program $p_{2(a)}$, but leave it undefined in program $p_{2(b)}$. The value for `x` is undefined in program $p_{2(c)}$ but the final value of `y` is 1 in both semantics.

course, the program fails to terminate and the final values of both `x` and `y` are undefined.

Under Cartwright and Felleisen's semantics, when slicing on `y`, at the end of program $p_{2(c)}$ the infinite loop can be deleted since it has no effect in lazy semantics on the final value of `y`. This is exactly what happens using slicing algorithms such as Weiser's.

2.2.2 The Transfinite Semantics of Giacobazzi and Mastroeni

Giacobazzi and Mastroeni [19] argue that if a semantics is to be useful for modelling kinds of program manipulation such as slicing it should be be able to capture semantic information 'beyond infinite loops' . They use transfinite state traces of programs [29] and show the existence of such semantics using domain equations. They introduce a non-standard semantics, called *transfinite semantics* using a metric structure on their value domains. Transfinite semantics of a program is defined in terms of the set of all possibly transfinite computations: computations whose length can be any ordinal, finite or infinite.

Figure 3 gives an example: executing program $p_{3(a)}$ produces a concatenation of an infinite trajectory with a finite or infinite trajectory (depending on whether the second loop executes a finite or infinite number of times). In program $p_{3(b)}$, we have an infinite concatenation of finite or infinite trajectories (the inner loop may execute any number of times on each iteration of the outer loop.).

2.2.3 The Semantics of Nestra

Nestra [35] defines a new transfinite semantics based on the semantics of Giacobazzi and Mastroeni [19]. Unlike the semantics of Giacobazzi and Mastroeni [19] where all the states observed during the infinite execution are used, Nestra shows that it suffices to consider a subset of these states. Nestra shows that

```
while( true ){
  a = a + 1;
}
while( y != z ){
  y = y - 1;
}


x = 2;
```

(a) Program $p_{3(a)}$

```
while( true ){
  a = a + 1;


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

(b) Program $p_{3(b)}$

Fig. 3. The transfinite trajectories approach of Giacobazzi and Mastroeni. In program $p_{3(a)}$ there is the concatenation of an infinite trajectory with a finite or infinite trajectory (depending on whether the second loop executes a finite or infinite number of times). In program $p_{3(b)}$ we have an infinite concatenation of finite or infinite trajectories (the inner loop may execute any number of times on each iteration of the outer loop.)

```
while( true ) {
  x = 0;
  x = 1;
}
```

(a) Program $p_{4(a)}$

```
while( true ) {


  x = 1;
}
```

(b) Program $p_{4(b)}$

Fig. 4. In the semantics of Giacobazzi and Mastroeni, the final value of the variable x in $p_{4(a)}$ is undefined but in $p_{4(b)}$ it is 1. In the semantics of Nestra, however, the final value of x is 1 in both cases.

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. When using the semantics of Giacobazzi and Mastroeni [19], the final value of the variable x when executing program $p_{4(a)}$ is undefined but in $p_{4(b)}$ it is 1. The semantics of Nestra [35], however, the final value of x is 1 in both cases.

3 The Shortcomings of Previous Approaches

In this section we describe the problems relating to the approaches described in Section 2.

3.1 Problems with Weiser's Finite Trajectory Semantics

The trajectories in Definition 2.6 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 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).

Slicing algorithms however, can be applied to *any* program; not just to programs which terminate in all states but also to programs whose termination conditions are not known. In these cases, we cannot guarantee that static backward equivalence is preserved. This is a major shortcoming of Weiser's semantics.

3.2 Problems with the Lazy and Transfinite Approaches

In this subsection, we highlight three problems relating to the lazy and transfinite approaches:

(1) the lack of substitutivity,
(2) the problems of undefined control, and
(3) their counter-intuitive nature.

3.2.1 Lack of Substitutivity

Substitutivity [27,28,39,43], simply means that we can substitute a part of a program by a semantically equivalent part and still preserve the semantics of the original program. It is a very natural property required of a semantics [15,20,21], especially if the semantics is to be used for arguing about the correctness of program transformations such as slicing. (In Theorem 5.1 we show that the finite trajectory semantics introduced in this paper is, indeed, substitutive.)

Danicic et al. [15], however, show that neither the semantics of Giacobazzi and Mastroeni [19] nor that of Cartwright and Felleisen [10] is substitutive. This is illustrated in Figure 5 by the substitution on line 6. In program $p_{5(a)}$, 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

```
1   x = 1;
2   y = 1;
3   while( true ) {
4      x = -x;
    }
5   if( x > 0 ) {
6      y = y;
7   } else {
8      x = -x;
    }
```

(a) Program, $p_{5(a)}$

```
1   x = 1;
2   y = 1;
3   while( true ) {
4      x = -x;
    }
5   if( x > 0 ) {
6      skip;
7   } else {
8      x = -x;
    }
```

(b) Program $p_{5(b)}$

Fig. 5. The programs $p_{5(a)}$ and $p_{5(b)}$ are not equivalent under either Cartwright and Felleisen's lazy semantics, under Giacobazzi and Mastroeni's transfinite semantics, or under Nestra's transfinite semantics. For program $p_{5(a)}$ both semantics give the final value of `y` as 'undefined'. For program $p_{5(b)}$ they both give the final value of `y` to be 1. However, the statements `y=y` and `skip` are semantically equivalent. Therefore none of these semantics is substitutive.

value of `y` is undefined when using either the semantics of Giacobazzi and Mastroeni [19], that of Nestra [35], or that of Cartwright and Felleisen [10]. Program $p_{5(b)}$ in Figure 5(b) is obtained by replacing the statement `y=y;` on line 6 of $p_{5(a)}$ with the semantically equivalent `skip`. However, in program $p_{5(b)}$, the semantics of Giacobazzi and Mastroeni [19], the semantics of Nestra [35], and the semantics of Cartwright and Felleisen [10] 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 corrected by Danicic et al. [15] 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 models the behaviour of Weiser's slicing algorithm for slicing criteria whose label is the end of a program, it does not cater for slicing at arbitrary points in the middle of the program.

3.2.2 Problems with Undefined Control

The transfinite computation approach appears to encounter problems when flow after an infinite loop depends on values computed by an infinite loop.

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


x = 2;
```

(a) Program $p_{6(a)}$

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


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

(b) Program $p_{6(b)}$

Fig. 6. The transfinite trajectories approach of Giacobazzi and Mastroeni. Examples where the number of iterations of a loop depend upon a previously undefined value.

The trajectory of programs like:

$$\texttt{while}(\text{true})\ x = x + 1;\ \ \texttt{if}\ p(x)\ p_1\ \texttt{else}\ p_2$$

does not appear to have been considered. After execution of the infinite loop the final value of variable x is undefined. Therefore we do not know whether p_1 or p_2 will be executed.

If both are allowed, we would have to consider the semantics of program to be a *set* of trajectories, not a single trajectory as implied by Giacobazzi and Mastroeni. In the case of the programs in Figure 5 each initial state would produce *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.

Using sets of trajectories to correct the problem of values undetermined after an infinite loop leads to more problems. For example, in program $p_{6(a)}$ in Figure 6, the set of trajectories 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. We suspect the problem of having to 'match up' which of these trajectories would have to agree when slicing would be difficult.

Matters are even more complex for program $p_{6(b)}$ in Figure 6 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.

3.2.3 The Semantics is Counter-Intuitive

Program semantics [41] is simply a mapping from the set of all programs to some co-domain: another set of mathematical objects. Trivial semantics can, of course, be defined; for example, a semantics that maps all program to the same object, or a semantics that maps every program to itself. These semantics, although well-defined, are not useful because they do not give us any insights into the behaviour of the programs they represent. Similarly, a semantics which given a slicing criterion, (V, l), simply maps every program to the result of applying Weiser's algorithm to it at (V, l) is well-defined. Although, by definition, it captures Weiser's algorithm more precisely than any other, it does not give us any insights into the behaviour preserved by the algorithm. For a semantics to be useful, its co-domain must correspond to some intuition about how the program, at least conceptually, executes. For example, in standard semantics, the co-domain corresponds to state-to-state mappings, where the state is a mapping representing the current values of all the variables. A problem with the lazy and transfinite approaches is that they define a semantics of programs that does not correspond to the actual execution of a program on a computer. It is a conceptual model that requires the reader, for example, to imagine programs continuing to execute after executing infinite loops (and even to imagine the execution of an infinite loop an infinite number of times). The fact that a slice preserves such a counter-intuitive semantics means that it is very difficult for the user of a slicing algorithm to envisage the connection between the original program and its slice. Clearly, for a semantics to be useful as a way of understanding this relationship, it is preferable if it is defined in terms of the 'normal' execution of programs on a von Neumann machine, i.e., using standard strict semantics.

In the next section we introduce a new semantics which overcomes these problems. It is a natural extension of the original semantics introduced by Weiser [46].

4 The New Trajectory Semantics

In this section, we introduce the new semantics, $\vec{\mathcal{T}}$ that is defined over *all* programs which extends the previously defined semantics $\mathcal{T}$, which was just defined for terminating programs. We first define our simple language with labels, sub-programs, and quotient programs:

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

where l denotes a label, e denotes an arithmetic expression and b denotes a boolean expression. We call an element of this language a *program*, and identify Γ with the set of all programs. The *sub-programs* of a program p are p itself and the sub-programs of all its components Γ_i, above.

Let n be a sub-program of p. If we replace n 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 [40,41], 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.

4.1 Standard Trajectory Semantics

Before we define our new semantics, $\vec{\mathcal{T}}$, we first define standard trajectory semantics, $\mathcal{T}$. This is the intuitive trajectory semantics where we allow trajectories to be infinite (but not transfinite) in order to handle non-termination. Terminating programs will give rise to finite trajectories and non-terminating programs to countably infinite trajectories in the *natural* way.

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

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

where Γ is the set of all programs, Σ is the set of all states, and $\mathrm{Seq}(L \times \Sigma)$ is 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 $p; q$ (where q is an arbitrary program) will be equal to the trajectory of p in state σ, since q is never reached. 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 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 effect, 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:
$$\mathcal{T}[\![\texttt{if}(l : b)\;\; p \texttt{ else } q]\!]\sigma = \langle(l, \sigma)\rangle \oplus (\mathcal{E}[\![b]\!]\sigma \rightarrow \mathcal{T}[\![p]\!]\sigma, \mathcal{T}[\![q]\!]\sigma)$$
where $(\mathit{True} \rightarrow a, b)$ is a, $(\mathit{False} \rightarrow a, b)$ is b, and $(\perp \rightarrow a, b)$ 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

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

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

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_i ($i \in \{2(a), 2(b), 2(c)\}$) in Figure 2 has trajectory

$$\mathcal{T}[\![p_i]\!]\sigma = \langle(1,\sigma)(2,\sigma_1)(1,\sigma_1)(2,\sigma_2)(1,\sigma_2)(2,\sigma_3)\ldots\rangle$$

for all initial states σ where, for all i, $\sigma_i = \sigma[x \leftarrow (\sigma(x)+i)]$. The trajectories through program $p_{6(b)}$ in Figure 6 depend on the initial state. Putting $\sigma = \{x \mapsto 1, y \mapsto 1, z \mapsto 0\}$ gives the following trajectory:

$$\mathcal{T}[\![p_{6(b)}]\!]\sigma = \langle(1,\sigma)(2,\sigma')(3,\sigma')(4,\sigma)(3,\sigma)(4,\sigma'')(3,\sigma'')(1,\sigma'')\ldots\rangle$$

where $\sigma' = \{x \mapsto 1, y \mapsto 2, z \mapsto 0\}$ and $\sigma'' = \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}$.

4.2 The New 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 is that $\vec{\mathcal{T}}_n[\![p]\!]\sigma$ will be finite for all n, p, σ. 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.

Purely, as an aid to giving some intuition to our semantics $\vec{\mathcal{T}}$, we introduce the notional n-machine: it is simply a 'computing engine' in which all loops are forced to terminate after n iterations. All programs terminate when executed on an n-machine. $\vec{\mathcal{T}}_n[\![p]\!]\sigma$ defines the semantics of executing program p in state σ on an n-machine.

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 unfolding does not remove the loops from the body.

Definition 4.1 (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:

$$\begin{aligned} W_0(l,b,p) &= \texttt{if}\ (l : b)\ \ \texttt{skip else skip} \\ W_{n+1}(l,b,p) &= \texttt{if}\ (l : b)\ \ p;\ W_n(l,b,p)\ \texttt{else skip} \end{aligned}$$

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:
$$\vec{\mathcal{T}}_n[\![\texttt{if}(l : b)\ \ p\ \texttt{else}\ q]\!]\sigma = \langle (l,\sigma)\rangle \oplus (\mathcal{E}[\![b]\!]\sigma \to \vec{\mathcal{T}}_n[\![p]\!]\sigma, \vec{\mathcal{T}}_n[\![q]\!]\sigma)$$
- 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
$$\vec{\mathcal{T}}^i_n[\![\texttt{while}(l : b)\ p]\!]\sigma = \\ \langle l,\sigma\rangle \oplus \begin{cases} \vec{\mathcal{T}}_n[\![p]\!]\sigma \oplus \vec{\mathcal{T}}^{i+1}_n[\![\texttt{while}(l : b)\ p]\!]\sigma' & \text{if } \mathcal{E}[\![b]\!]\sigma \text{ and } i \leqslant n, \text{ or} \\ \langle\rangle & \text{otherwise.} \end{cases}$$
where, again, σ' is the state component of the last element of $\vec{\mathcal{T}}_n[\![p]\!]\sigma$.

For example, for program $p_{2(a)}$ in Figure 2 we obtain the finite trajectories

$$\begin{aligned}
\vec{\mathcal{T}}_0[\![p_{2(a)}]\!]\sigma &= \langle(1,\sigma)(3,\sigma[x \leftarrow 1])\rangle\\
\vec{\mathcal{T}}_1[\![p_{2(a)}]\!]\sigma &= \langle(1,\sigma)(2,\sigma_1)(1,\sigma_1)(3,\sigma[x \leftarrow 1])\rangle\\
\vec{\mathcal{T}}_2[\![p_{2(a)}]\!]\sigma &= \langle(1,\sigma)(2,\sigma_1)(1,\sigma_1)(2,\sigma_2)(1,\sigma_2)(3,\sigma[x \leftarrow 1])\rangle\\
&\ \vdots
\end{aligned}$$

where, again, for all i, $\sigma_i = \sigma[x \leftarrow (\sigma(x) + i)]$.

The trajectories through program $p_{6(b)}$ in Figure 6, on the other hand, depend on the initial state. Let $\sigma_i = \{x \mapsto 1, y \mapsto i, z \mapsto 0\}$, for all i, then:

$$\begin{aligned}
\vec{\mathcal{T}}_0[\![p_{6(b)}]\!]\sigma_1 &= \langle(1,\sigma_1)(5,\sigma_1[x \leftarrow 2])\rangle\\
\vec{\mathcal{T}}_1[\![p_{6(b)}]\!]\sigma_1 &= \langle(1,\sigma_1)(2,\sigma_2)(3,\sigma_2)(4,\sigma_1)(3,\sigma_1)(1,\sigma_1)(5,\sigma_1[x \leftarrow 2])\rangle\\
\vec{\mathcal{T}}_2[\![p_{6(b)}]\!]\sigma_1 &= \langle(1,\sigma_1)(2,\sigma_2)(3,\sigma_2)(4,\sigma_1)(3,\sigma_1)(4,\sigma_0)(3,\sigma_0)\\
&\qquad (1,\sigma_0)(2,\sigma_1)(3,\sigma_1)(4,\sigma_0)(3,\sigma_0)(5,\sigma_0[x \leftarrow 2])\rangle\\
&\ \vdots
\end{aligned}$$

Notice that in $\vec{\mathcal{T}}_0[\![p_{6(b)}]\!]\sigma_1$ the loop unfolding forces the outer loop to exit immediately. In $\vec{\mathcal{T}}_1[\![p_{6(b)}]\!]\sigma_1$ both loops are forced to exit early. However in $\vec{\mathcal{T}}_n[\![p_{6(b)}]\!]\sigma_1$ for $n \geqslant 2$ the inner loop exits naturally and only the outer loop is forced to exit.

4.3 Finite Trajectory Backward Slice Equivalence

In this section, having defined $\vec{\mathcal{T}}$, we are now in a position to define the new equivalence relation: *finite trajectory backward slice equivalence*, $\vec{\mathcal{S}}_{(V,l)}$, on programs. We prove that $\vec{\mathcal{S}}_{(V,l)}$ is, indeed, an equivalence relation and prove importantly, that it is a natural extension of $\mathcal{S}_{(V,l)}$. This means that for always-terminating programs, $\vec{\mathcal{S}}_{(V,l)}$ and $\mathcal{S}_{(V,l)}$ are identical relations.

Finite trajectory backward slice equivalence can be defined intuitively in terms of n-machines: Programs p and q are *finite trajectory backward slice equivalent* with respect to slicing criterion (V, l) if for all states σ, there exists a sufficiently large N^σ, such that for all larger n, when run on an n-machine they will both pass through l the same number of times and the values of all the variables in V will agree at corresponding times through l.

Definition 4.2 (Finite trajectory backward slice equivalence) *Let $p, q \in \Gamma$ be programs, and let (V, l) be a slicing criterion. Program p is* static backward equivalent *to q if and only if*

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

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

Lemma 4.1 *If (V,l) is a slicing criterion then $\vec{\mathcal{S}}_{(V,l)}$ is an equivalence relation.*

Proof. To prove reflexivity and symmetry is trivial. We now prove transitivity. Let $p, q, r \in \Gamma$ be programs and (V,l) be a slicing criterion, and suppose that $p\ \vec{\mathcal{S}}_{(V,l)}\ q$ and $q\ \vec{\mathcal{S}}_{(V,l)}\ r$, i.e.,

$$\begin{aligned} \forall \sigma \in \Sigma : \exists N_1^\sigma \in \mathbb{N} : \forall m \geqslant N_1^\sigma : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![p]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![q]\!]\sigma) \\ \text{and } \forall \sigma \in \Sigma : \exists N_2^\sigma \in \mathbb{N} : \forall m \geqslant N_2^\sigma : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![q]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![r]\!]\sigma). \end{aligned}$$

Choose $N^\sigma \geqslant \max(N_1^\sigma, N_2^\sigma)$ to give

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

from which it follows that $\forall m \geqslant N^\sigma$:

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

Lemma 4.2 If p is a program that halts on input state σ then $\exists N^\sigma \in \mathbb{N}$: $\forall m \geqslant N^\sigma : \vec{\mathcal{T}}_m[\![p]\!]\sigma = \mathcal{T}[\![p]\!]\sigma$.

Proof. Given an input state σ on which p halts, the trajectory $\mathcal{T}[\![p]\!]\sigma$ is finite. Choose a sufficiently large N^σ that is greater than the number of any iterations of any loops we encounter during the execution of p on σ. In this case, the definition of $\vec{\mathcal{T}}_{N^\sigma}[\![p]\!]$ reduces to the definition $\mathcal{T}[\![p]\!]$, since the only difference in these definitions are the handling of `while` loops.

Since N^σ is sufficiently large, the $i \leqslant N^\sigma$ condition in the definition of $\vec{\mathcal{T}}^i_{N^\sigma}[\![\texttt{while}(l : b)\ S]\!]$ is always true, which makes the difference between the two definitions disappear. Thus, for the chosen N^σ it holds that $\forall m \geqslant N^\sigma$: $\vec{\mathcal{T}}_m[\![p]\!]\sigma = \mathcal{T}[\![p]\!]\sigma$. $\square$

We now show that our new semantic equivalence $\vec{\mathcal{S}}_{(V,l)}$ agrees completely with static backward equivalence when applied to terminating programs. In other words, $\vec{\mathcal{S}}_{(V,l)}$ is a natural extension to non-terminating programs of $\mathcal{S}_{(V,l)}$.

Theorem 4.1 For all always-terminating programs p and q and slicing criteria (V,l), $p\ \mathcal{S}_{(V,l)}\ q$ if and only if $p\ \vec{\mathcal{S}}_{(V,l)}\ q$.

Proof. By Definitions 2.6 and 4.2 we have to show that for all slicing criteria (V, l), for all always-terminating programs p and q, and for all input states σ,

$$\mathrm{Proj}_{(V,l)}(\mathcal{T}[\![p]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\mathcal{T}[\![q]\!]\sigma)$$
$$\text{if and only if}$$
$$\exists N^\sigma \in \mathbb{N} : \forall m \geqslant N^\sigma : \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![p]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_m[\![q]\!]\sigma).$$

<u>Only if:</u> Since we know that p always terminates, by Lemma 4.2 we also know that there is an N_p^σ such that for all $m \geqslant N_p^\sigma$, $\mathcal{T}[\![p]\!]\sigma = \vec{\mathcal{T}}_m[\![p]\!]\sigma$. Similarly, we know that an N_q^σ exists such that for all $m \geqslant N_q^\sigma$, $\mathcal{T}[\![q]\!]\sigma = \vec{\mathcal{T}}_m[\![q]\!]\sigma$. Let N^σ be the maximum of N_p^σ and N_q^σ. Then, for all $m \geqslant N^\sigma$ it still holds that $\mathcal{T}[\![p]\!]\sigma = \vec{\mathcal{T}}_m[\![p]\!]\sigma$ and $\mathcal{T}[\![q]\!]\sigma = \vec{\mathcal{T}}_m[\![q]\!]\sigma$. Inserting these in our original assumption of $\mathrm{Proj}_{(V,l)}(\mathcal{T}[\![p]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\mathcal{T}[\![q]\!]\sigma)$, this yields for the chosen N^σ

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

<u>If:</u> Now, our assumption is that

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

We also know that there is an N_p^σ such that $\forall m \geqslant N_p^\sigma : \mathcal{T}[\![p]\!]\sigma = \vec{\mathcal{T}}_m[\![p]\!]\sigma$ and that there is an N_q^σ such that $\forall m \geqslant N_q^\sigma : \mathcal{T}[\![q]\!]\sigma = \vec{\mathcal{T}}_m[\![q]\!]\sigma$.

Let N'^σ be the maximum of N^σ, N_p^σ and N_q^σ. It follows that $\mathcal{T}[\![p]\!]\sigma = \vec{\mathcal{T}}_{N'^\sigma}[\![p]\!]\sigma$, $\mathcal{T}[\![q]\!]\sigma = \vec{\mathcal{T}}_{N'^\sigma}[\![q]\!]\sigma$, and $\mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_{N'^\sigma}[\![p]\!]\sigma) = \mathrm{Proj}_{(V,l)}(\vec{\mathcal{T}}_{N'^\sigma}[\![q]\!]\sigma)$, from which it follows that

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

Thus the new equivalence $\vec{\mathcal{S}}_{(V,l)}$ is a natural extension of the old equivalence, $\mathcal{S}_{(V,l)}$. $\vec{\mathcal{S}}_{(V,l)}$ however, also applies to programs which do not always terminate and is hence defined for a much larger class of programs.

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

In this section we prove our main result (Theorem 4.2), that slicing algorithms which use data and control dependence satisfy the new 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 defined at the beginning of this section, control dependence can be defined in terms of the program structure.

Definition 4.3 (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 4.4 (Finite syntactic paths) *For any program p, the set of finite syntactic paths of p, denoted by $\mathcal{P}(p)$, is defined as follows:*

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

Here, $\oplus$ is concatenation and * is Kleene closure. 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 4.5 (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 $\emptyset \neq \operatorname{def}(l) \subseteq \operatorname{ref}(l')$ and for all $l'' \in \pi_2$, $\operatorname{def}(l'') \neq \operatorname{def}(l)$. Each π_i may be an empty path, $\operatorname{def}(l)$ is the set of variables defined at l (a set of maximum one element), and $\operatorname{ref}(l)$ denotes the set of variables referenced at l.*

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

Definition 4.6 (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 [25] where there is a desire to slice on a criterion (V, l) that does not satisfy this a new node – an 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') 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 4.6 (the slice produced by Weiser's and the PDG algorithm) then $p\ \vec{\mathcal{S}}_{(V,l)}\ q$.

We generalise Definition 2.3 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 2.4 to give

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

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

We now are in a position to prove Lemma 4.3, from which our main result, Theorem 4.2, follows immediately. Observe that the statement of Lemma 4.3 is stronger than necessary, since we prove the condition for all $n \in \mathbb{N}$ whereas, in fact, it would have been sufficient to prove that the condition holds for n sufficiently large.

Lemma 4.3 *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.

[3] Weiser handled this problem by slicing with respect to the 'nearest successor' of the slice point that is in the slice [46].

Proof. Write

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

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

$$\begin{aligned}\mathrm{Proj}_L(\vec{\mathcal{T}}_m[\![p]\!]\sigma) &= (l_{f(1)}, \sigma_{f(1)}\!\downarrow\!\mathrm{def}(l_{f(1)})) \dots (l_{f(i)}, \sigma_{f(i)}\!\downarrow\!\mathrm{def}(l_{f(i)})) \qquad (1)\\ \text{and } \mathrm{Proj}_L(\vec{\mathcal{T}}_m[\![q]\!]\sigma) &= (l'_{g(1)}, \sigma'_{g(1)}\!\downarrow\!\mathrm{def}(l'_{g(1)})) \dots (l'_{g(h)}, \sigma'_{g(h)}\!\downarrow\!\mathrm{def}(l'_{g(h)})) \qquad (2)\end{aligned}$$

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 $l'' \xrightarrow[p]{\text{data}} l_{f(t)}$ 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 $l_{f(t)} \neq l'_{g(t)}$ 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 4.2 *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. The fact that

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

follows immediately from Lemma 4.3.

We have, thus, proved that dependence-based slices satisfy our new equivalence $\vec{\mathcal{S}}_{(V,l)}$ for every slicing criterion (V, l). □

5 Comparison with Previous Approaches

In this section, we compare our approach with the other semantics discussed in Section 2. Unlike the lazy and transfinite approaches [10,19,35], $\vec{\mathcal{T}}$ is substitutive. In Section 3.2.2, we highlighted a problem with the transfinite approach: the semantics is not defined when the program control flow after an infinite loop depends on values computed by an infinite loop. Our semantics, however, is well-defined in such situations, since it only considers finite unfoldings. Finally, we argue that our approach is more intuitive than these approaches since it is based on standard semantics.

5.1 A Natural Extension of Static Backward Equivalence

The trajectories in Definition 2.6 are finite, hence static backward equivalence is only defined for programs which terminate. $\mathcal{S}_{(V,l)}$ 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).

Slicing algorithms however, are applied not just to programs which terminate in all states but also to programs whose termination conditions are not known. In these cases, we cannot guarantee that static backward equivalence is preserved. This is a major shortcoming of static backward equivalence.

Theorem 4.1 shows that our new semantic equivalence $\vec{\mathcal{S}}_{(V,l)}$, naturally extends static backward equivalence, $\mathcal{S}_{(V,l)}$ to cover both terminating and non-terminating programs.

5.2 $\vec{\mathcal{T}}$ is Substitutive

In this subsection, we prove that $\vec{\mathcal{T}}$ is substitutive.

Definition 5.1 (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.*

Theorem 5.1 ($\vec{\mathcal{T}}$ 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

$$\begin{aligned} \vec{\mathcal{T}}_n[\![p'_1; p'_2]\!]\sigma &= \vec{\mathcal{T}}_n[\![p'_1]\!]\sigma \oplus \vec{\mathcal{T}}_n[\![p'_2]\!]\sigma' && \text{(by definition)} \\ &= \vec{\mathcal{T}}_n[\![p_1]\!]\sigma \oplus \vec{\mathcal{T}}_n[\![p_2]\!]\sigma' && \text{(by induction)} \\ &= \vec{\mathcal{T}}_n[\![p_1; p_2]\!]\sigma \end{aligned}$$

  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

$$\begin{aligned} &\vec{\mathcal{T}}_n[\![\texttt{if}(l : b)\ \ p'_1 \ \texttt{else}\ p'_2]\!]\sigma \\ &\quad = \langle (l, \sigma) \rangle \oplus (\mathcal{E}[\![b]\!]\sigma \rightarrow \vec{\mathcal{T}}_n[\![p'_1]\!]\sigma, \vec{\mathcal{T}}_n[\![p'_2]\!]\sigma) && \text{(by definition)} \\ &\quad = \langle (l, \sigma) \rangle \oplus (\mathcal{E}[\![b]\!]\sigma \rightarrow \vec{\mathcal{T}}_n[\![p_1]\!]\sigma, \vec{\mathcal{T}}_n[\![p_2]\!]\sigma) && \text{(by induction)} \\ &\quad = \vec{\mathcal{T}}_n[\![\texttt{if}(l : b)\ \ p_1 \ \texttt{else}\ p_2]\!]\sigma \end{aligned}$$

- 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. $\square$

5.3 Finite Trajectory Semantics is More Intuitive

A problem with the lazy and transfinite approaches is that they define a semantics of programs that does not resemble the actual execution of a program on a computer. They are conceptual models that requires the reader, for example, to imagine programs continuing to execute after executing infinite loops (and even to imagine the execution of an infinite loop an infinite number of times). The fact that a slice preserves such a counter-intuitive semantics means that it is very difficult for the user of a slicing algorithm to envisage the connection between the original program and its slice. Clearly, for a semantics to be useful as a way understanding this relationship, it is preferable if it is defined in terms of the normal execution of programs on a computer, i.e., using standard strict semantics. This is true of finite trajectory backward slice equivalence, which has a comparatively intuitive definition:

An n-machine is a notional computing engine on which all loops are forced to terminate after n iterations. Programs p and q are *finite trajectory backward slice equivalent* with respect to slicing criterion (V, l) if for all states σ, there exists a sufficiently large N^{σ}, such that for all larger n, when run on an n-machine they will both pass through l the same number of times and the values of all the variables in V will agree at corresponding times through l.

6 Conclusions and Future Work

Motivated by the need to understand the behaviour of slicing algorithms when applied to reactive systems, we have introduced a theory for slicing non-terminating programs. In achieving this, our main contributions have been:

(1) to introduce a new, substitutive, intuitive, finite trajectory-based semantics, $\vec{\mathcal{T}}$, of programs (Section 4.2),
(2) to prove (Theorem 4.1) that its induced equivalence, $\vec{\mathcal{S}}_{(V,l)}$, is a natural extension, to non-terminating programs, of the equivalence, $\mathcal{S}_{(V,l)}$, introduced by Weiser [46] that only considers terminating programs and
(3) to prove (Theorem 4.2) that slicing algorithms based on traditional data and control dependence preserve $\vec{\mathcal{T}}$, thereby showing that the new semantics captures the behaviour of program slicing algorithms for both terminating and non terminating programs.

In this paper we have considered concrete programs, but it is well-known that program schemas [11,13,34] – equivalence classes of programs – contain all the necessary information to formally investigate dependence-based slicing algorithms. Future work will attempt to generalise the ideas in this paper by further investigating the relationship between slicing algorithms and schema theory.

We have focused on static backward slicing, and especially on the extension of the static backward equivalence relation $\mathcal{S}_{(V,l)}$ on terminating programs to $\vec{\mathcal{S}}_{(V,l)}$ on all programs. Our idea of limiting loops to a specific number of iterations (or, equivalently, unfolding them a specific number of times) can be applied to all other forms of slicing formally treated in our previous work [5,1,3,4]. Most importantly, it can be applied to the unified equivalence defined by Binkley et. al [5]. We will also investigate the relationship between the resulting (dynamic) definitions and the existing (dynamic) slicing algorithms.

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).

David 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] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Lahcen Ouarbya. Formalizing executable dynamic and forward slicing. In 4th *International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, pages 43–52, Los Alamitos, California, USA, September 2004. IEEE Computer Society Press.

[2] David Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11 and 12):583–594, 1998.

[3] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. Minimal slicing and the relationships between forms of slicing. In 5th *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 05)*, pages 45–54, Los Alamitos, California, USA, 2005. IEEE Computer Society Press. Best paper award winner.

[4] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. A formalisation of the relationship between forms of program slicing. *Science of Computer Programming*, 62(3):228–252, 2006.

[5] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. Theoretical foundations of dynamic program slicing. *Theoretical Computer Science*, 360(1):23–41, 2006.

[6] David Binkley and Keith Brian Gallagher. Program slicing. In Marvin Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[7] David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[8] David Binkley, Susan Horwitz, and Tom Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, 1995.

[9] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):595–607, 1998.

[10] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–27, 1989.

[11] Sebastian Danicic, Chris Fox, Mark Harman, Robert Mark Hierons, John Howroyd, and Mike Laurence. Slicing algorithms are minimal for programs which can be expressed as linear, free, liberal schemas. *The computer Journal*, 48(6):737–748, 2005.

[12] Sebastian Danicic, Mark Harman, Robert Hierons, John Howroyd, and Mike Laurence. Applications of linear program schematology in dependence analysis. In $1^{st.}$ *International Workshop on Programming Language Interference and Dependence*, Verona, Italy, August 2004.

[13] Sebastian Danicic, Mark Harman, Robert Mark Hierons, John Howroyd, and Mike Laurence. Equivalence of linear, free, liberal, structured program schemas is decidable in polynomial time. *Theoretical Computer Science*, 373:1–18, March 2007.

[14] Sebastian Danicic, Mark Harman, John Howroyd, and Lahcen Ouarbya. A lazy semantics for program slicing. In $1^{st.}$ *International Workshop on Programming Language Interference and Dependence*, Verona, Italy, August 2004.

[15] Sebastian Danicic, Mark Harman, John Howroyd, and Lahcen 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*, 72:123–240, July-August 2007.

[16] Andrea De Lucia. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Los Alamitos, California, USA, 2001. IEEE Computer Society Press.

[17] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[18] Keith B. Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[19] Roberto Giacobazzi and Isabella Mastroeni. Non–standard semantics for program slicing. *Higher-Order and Symbolic Computation*, 16(4):297–339, 2003. Special issue on Partial Evalution and Semantics-Based Program Manipulation.

[20] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, October 2003.

[21] Mark Harman and Sebastian Danicic. Amorphous program slicing. In 5th *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, pages 70–79, Los Alamitos, California, USA, May 1997. IEEE Computer Society Press.

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

[23] Mark Harman, Lin Hu, Robert Mark Hierons, Joachim Wegener, Harmen Sthamer, Andr Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, January 2004.

[24] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, December 2000.

[25] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[26] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[27] Michael Huth. On the approximation of denotational mu-semantics. *Applied Categorical Structures*, pages 1–27, 1998.

[28] Cezary Kaliszyk, Pierre Corbineau, Freek Wiedijk, James McKinna, and Herman Geuvers. A real semantic web for mathematics deserves a real semantics. In *SemWiki*, 2008.

[29] J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. Vries. Transfinite reduction in orthogonal term rewriting systems. *Information and computation*, 119(1):18–38, 1995.

[30] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 155–169, N.Y., January 19–21 2000. ACM Press.

[31] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[32] Bogdan Korel, Inderdeep Singh, Luay Tahat, and Boris Vaysburg. Slicing of state based models. In *IEEE International Conference on Software Maintenance (ICSM'03)*, pages 34–43, Los Alamitos, California, USA, September 2003. IEEE Computer Society Press.

[33] Arun Lakhotia and P. Singh. Challenges in getting formal with viruses. *virus bulletin*, September 2003.

[34] Michael R. Laurence, Sebastian Danicic, Mark Harman, Rob Hierons, and John Howroyd. Equivalence of conservative, free, linear program schemas is decidable. *Theoretical Computer Science*, 290:831–862, January 2003.

[35] Härmel Nestra. Transfinite semantics in program slicing. *Proc. Estonian Acad. Sci. Eng.*, 11(4):313–328, Dec 2005.

[36] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, and John Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *European Symposium on Programming*, pages 77–93, 2005.

[37] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007.

[38] Thomas Reps and Wuu Yang. The semantics of program slicing. Technical Report Technical Report 777, University of Wisconsin, 1988.

[39] David Sands. A compositional semantics of combining forms for gamma programs. In *Formal Methods in Programming and Their Applications*, pages 43–56, 1993.

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

[41] Joseph E. Stoy. *Denotational semantics: The Scott–Strachey approach to programming language theory.* MIT Press, 1985. Third edition.

[42] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[43] Janis Voigtländer and Patricia Johann. Selective strictness and parametricity in structural operational semantics, inequationally. *Theor. Comput. Sci.*, 388(1-3):290–318, 2007.

[44] Martin Ward. Program slicing via FermaT transformations. In 26th *IEEE Annual Computer Software and Applications Conference (COMPSAC 2002)*, pages 357–362, Los Alamitos, California, USA, August 2002. IEEE Computer Society Press.

[45] Martin Ward and Hussein Zedan. Slicing as a program transformation. *ACM Trans. Program. Lang. Syst.*, 29(2):7, 2007.

[46] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[47] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[48] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.